NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

SERVER PROBING FOR SERVER AND AGENT BASED ACTIVE NETWORK MANAGEMENT

by

Mustafa Altinkaya

March 2000

Thesis Advisor:	Geoffrey Xie
Second Reader:	James Bret Michael

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
March 2000.

3. REPORT TYPE AND DATES COVERED
Master's Thesis

4. TITLE AND SUBTITLE Server Probing for Server and Agent Based Active Network Management

5. FUNDING NUMBERS

6. AUTHOR(S) Mustafa Altinkaya

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
DARPA and NASA

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
G417

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

In Server and Agent Based Active Network Management (SAAM) architecture, a server will make routing and other important decisions on behalf of the routers in its region. In order to make the right decisions and to support QoS (e.g., IntServ and DiffServ), the SAAM server needs to maintain an accurate region-wide view of network performance. This will be achieved as routers periodically send Link State Advertisement (LSA) messages to the SAAM server. Currently, the LSA messages report two key Link Performance Statistics, the average delay and the loss rate experienced by packets. Moreover, the server needs to perform sanity checks of these statistics by probing specific links. This thesis describes a server probing solution in which the SAAM server probes a router by dynamically injecting customized probing programs into the adjacent routers. In other words, the probing will be done with the active networking approach. An important feature of the server probing solution is that the probing activities cannot be detected by the router being probed.

14. SUBJECT TERMS Active networking, Quality of Service, Networks.

15. NUMBER OF PAGES 196

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
Unclassified

18. SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

19. SECURITY CLASSIFICATION OF ABSTRACT
Unclassified

20. LIMITATION OF ABSTRACT
UL

THIS PAGE INTENTIONALLY LEFT BLANK

SERVER PROBING FOR SERVER AND AGENT BASED ACTIVE NETWORK MANAGEMENT

Mustafa Altinkaya
Lieutenant Junior Grade, Turkish Navy
B.S., Turkish Naval Academy, 1994

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2000

Author: ______________________
Mustafa Altinkaya

Approved by: ______________________
Geoffrey Xie, Thesis Advisor

James Bret Michael
____
James Bret Michael, Second Reader

Dan C Boger
____
Dan Boger, Chairman
Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

In Server and Agent Based Active Network Management (SAAM) architecture, a server will make routing and other important decisions on behalf of the routers in its region. In order to make the right decisions and to support QoS (e.g., IntServ and DiffServ), the SAAM server needs to maintain an accurate region-wide view of network performance. This will be achieved as routers periodically send Link State Advertisement (LSA) messages to the SAAM server. Currently, the LSA messages report two key Link Performance Statistics, the average delay and the loss rate experienced by packets. Moreover, the server needs to perform sanity checks of these statistics by probing specific links. This thesis describes a server probing solution in which the SAAM server probes a router by dynamically injecting customized probing programs into the adjacent routers. In other words, the probing will be done with the active networking approach. An important feature of the server probing solution is that the probing activities cannot be detected by the router being probed.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGEMENTS

I would like to thank Dr. Geoffrey Xie for his assistance in my research. His guidance, support, knowledge and enthusiasm helped me to carry this thesis to completion. His efforts to provide and access resources for my study were invaluable. I would also like to thank my second reader, James Bret Michael, for his support.

I express my deepest gratitude to my family, whose love and patience have been a constant inspiration to me throughout my two years of study at the Naval Postgraduate School.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. THE INTERNET

The Internet dates back to the late 60's, that started as a project named "ARPANET." Mainly experimental, ARPANET was used for academic research purposes, and one of its main goals was to tolerate link failures. The Transmission Control Protocol/Internet Protocol (TCP/IP) suite was developed for this purpose, and was accepted as the standard protocol for the ARPANET.

The network that began as ARPANET has become the global network what we know as the "Internet." Nobody can precisely estimate the number of hosts or users connected to the Internet today, because no central authority exists that controls the entire network. However, there is no doubt that the Internet is growing at an explosive rate. In 1983, approximately 500 hosts were connected to the network. This number expanded to over 300,000 hosts in 1990, and to over 5 million in 1995.

The use of the Internet has also changed since the early days. Five to seven years ago, the Internet was mainly used for file transfer, e-mail, and access to the World Wide Web. Nowadays the transmission of voice and video are prevalent on the Internet. These types of data require certain Quality of Service (QoS) from the network.

The increase in volume and diversity of the network traffic raises new challenges for network management. In order to support applications as diverse as teleconferencing, video-on-demand, e-commerce and distributed computing, the resource management of the Internet needs to be improved.

B. QUALITY OF SERVICE

1. Voice and Video Requirements Over IP

Transmitting audio or video over a network requires more QoS guarantees from the network than just sufficient bandwidth. These types of applications are sensitive to timeliness of data; and hence we call such applications "real-time applications." Real-time applications need a guarantee from the network that their data will arrive on time and smoothly.

For example, to support high quality audio conferencing, the network needs to provide a low-latency and low-jitter service. Latency measures the time for a packet to travel from source to destination, and jitter measures the variation in latency for a set of packets. On the other hand non-real-time applications just need to make sure that their data arrives correctly. In this situation, a "best-effort" type of delivery in which the network just tries its best to deliver the packets but doesn't give any guarantee is sufficient.

One might think that more bandwidth would meet the requirements of voice and video. Improving the infrastructure of the Internet with fiber optic data links is one aspect of providing more bandwidth. However, in the short term, this solution would be neither feasible nor cheap. Applications are becoming more bandwidth-hungry, and that is true now and will remain true for the future. Better network management software for the Internet QoS capabilities, that would not require major change in the infrastructure is essential. The new system should also be backward compatible, meaning that the system should support the best-effort delivery.

2. IETF Proposals

There is a great deal of on-going research being done to resolve the lack of QoS for the Internet. In particular, the Internet Engineering Task Force (IETF) has proposed some service models and protocols for meeting the demand of QoS. Currently, the main service models and protocols are: Resource ReSerVation Protocol (RSVP), Integrated Services (IS), Differentiated Services (DS) and Multiprotocol Label Switching (MPLS) [Ref.5].

C. SERVER AND AGENT BASED ACTIVE NETWORK MANAGEMENT

Under the DARPA funded Next Generation Internet (NGI) initiative, a research project was initiated at Naval Postgraduate School to develop a Server and Agent Based Active Network Management (SAAM) system that provides an efficient solution for QoS. The SAAM architecture is designed to scale well with integrated services.

1. SAAM Architecture

In the current network architecture each stand-alone router needs to participate in almost all routing and management tasks --an inefficient way to meet the Quality of

Service (QoS) demands of the Next Generation Internet. The SAAM architecture will relieve the individual routers from the majority of routing and network management tasks. The proposed SAAM architecture consists of a SAAM server that controls a SAAM region including a number of routers (see Figure 2). The server will collect the global picture of the region and will make decisions on behalf of the routers. This method will provide a lightweight router that will perform only its primary task of forwarding the packets to their destination addresses. In this manner, SAAM allows the deployment of sophisticated software solutions to servers, the implementation of network QoS, and optimization of resources usage.

In order to illustrate the concept of SAAM, consider road traffic monitoring and control during commute hours in a big city like Istanbul. The radio stations are the management entities for the city's traffic control. In order to collect a global picture of the traffic in the city, radio stations send out helicopters to monitor the traffic on a particular region. The helicopters collect the information and send it back to the radio control station. Information obtained from different sources is aggregated, and the radio stations broadcast advice messages. In this way commuters can get the traffic information in "real-time". This early warning of congestion is the key for effective traffic control.

2. SAAM Server and Router

The SAAM Server will control a set of lightweight routers in terms of functionality, and will perform decision-making tasks for the routers. The SAAM Server will have two major functions.

First, the server will maintain the global view of the SAAM region by building a Path Information Base (PIB) to support QoS. With the PIB the server will be able to assign paths for flows that require QoS. The Server will identify the possible paths to route flows (see Figure 1). The server will also compute path performance parameters by collecting link performance data (Link State Advertisement) sent from the lightweight routers.

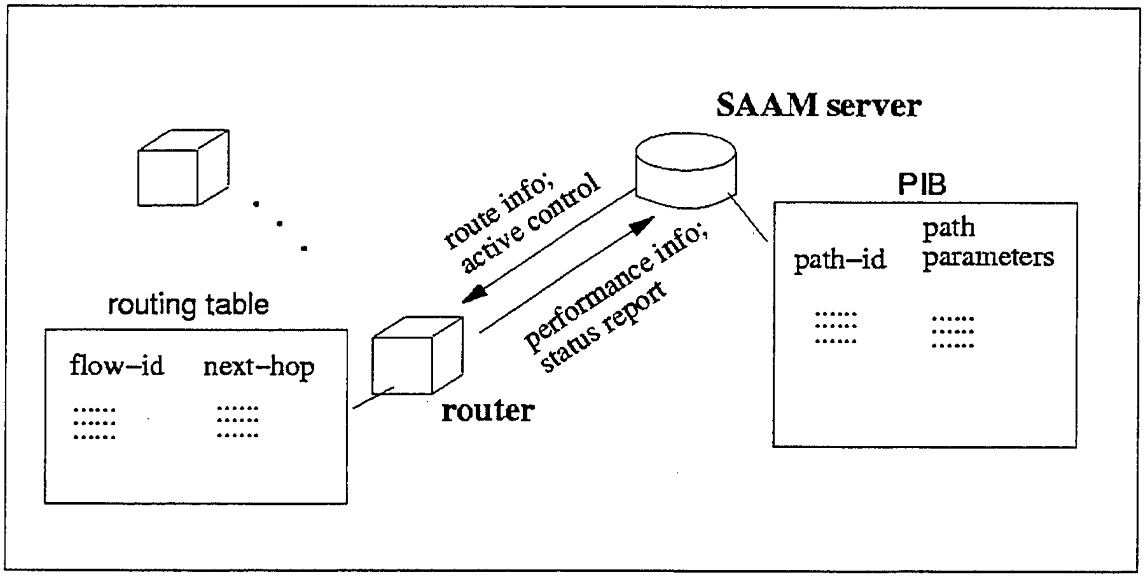


Figure 1. Logical Model of SAAM [From Ref. 1]

Second, the SAAM Server will make decisions on behalf of the routers in the region. Recall that each SAAM region will consist of at least one SAAM Server and a set of lightweight routers. The router will pass flow requests to the server. Once the server receives a flow request, it will compute the best route for that particular request, and pass routing table updates to the routers. The router is only responsible for routing packets.

3. Hierarchical Organization of the SAAM Servers

For scalability purposes, SAAM will organize its servers in a hierarchical model similar to the way in which the Domain Name Service (DNS) is organized. Each server will be responsible for a set of nodes. SAAM partitions the network into autonomous regions and sets up one server for each region. Each server will communicate with its immediate *children*, which might be one of the lightweight routers or a SAAM server. An example of a two-level server hierarchy is illustrated in Figure 2.

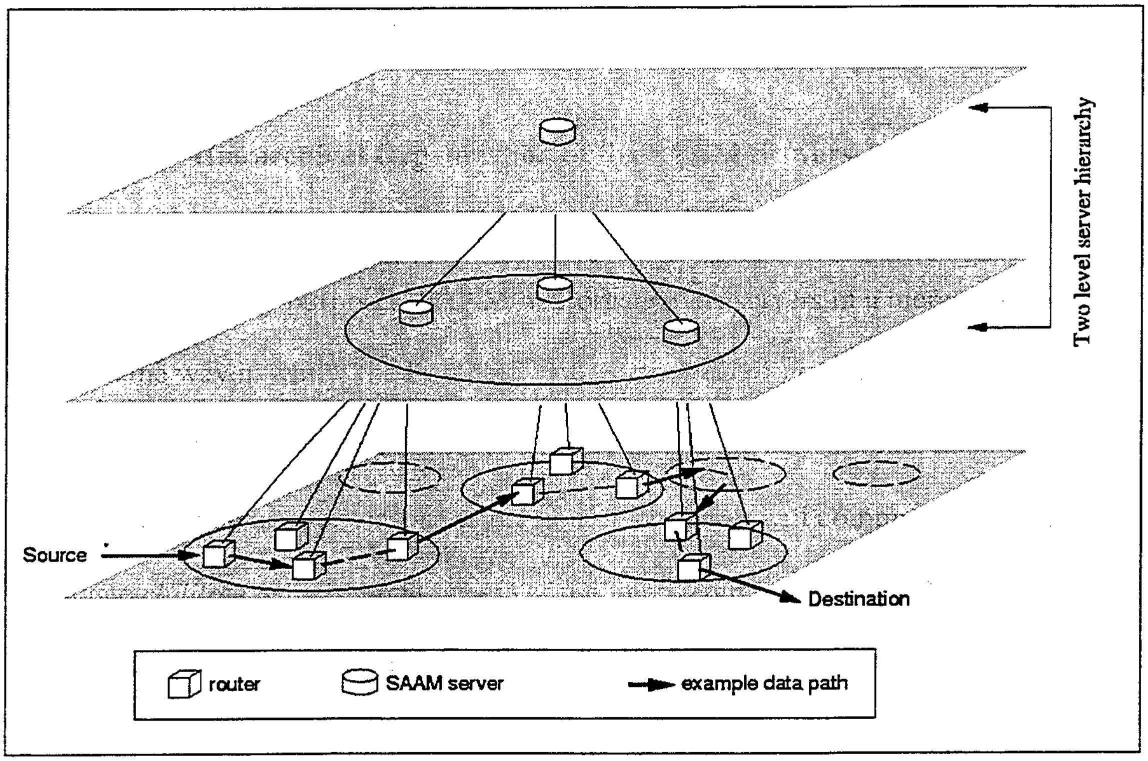


Figure 2. Hierarchical Organization of SAAM Servers [From Ref. 1]

The first level of the hierarchy controls a set of routers in a SAAM region. Each server needs to deal with flows in its region. If a flow needs to go across another region, then the second level SAAM Server needs manage this request. The first level servers will summarize the QoS parameters of its region for the second level server. Thus the second level server will have the big picture of the three SAAM regions as shown in Figure 2. This summary will allow the server to present itself as a router to the second level server.

In addition to being scalable, another important advantage of the hierarchical organization of the SAAM Servers is that it provides the gradual deployment of SAAM into today's network, and still compatible with the existing networks. As stated previously, the SAAM architecture improves the usage of the network resources without making a major change in the current system, and hence best effort traffic will still be supported by this system.

4. Benefits of SAAM

The Internet consists of many Internet Service Providers (ISP), which are operated independently. The primary concern of the ISPs is how to attract more customers to increase the ISP's total profits. More customers means more traffic. The ability to accommodate more traffic without changing the infrastructure would be very advantageous for the ISPs. SAAM meets such needs by improving the usage of the resources while providing more predictable performance to data flows. With SAAM, an ISP can provide several QoS levels to their customers including guaranteed, premium, assured or best effort. At the same time, a customer can decide what type of QoS level he or she needs, and pay according to that selection. The SAAM architecture will be backward compatible, meaning that it can be deployed gradually, and will not affect the existing infrastructure.

D. SCOPE OF THIS THESIS

The primary goal of this thesis is to add server performance probing capabilities to the SAAM prototype. This addition to the server will explicitly deploy code (executable modules) to particular links or nodes to measure throughput, loss rate, latency, and utilization. According to the received performance information, the server will verify whether or not the LSA (Link State Advertisement) reports from the routers under probe are acceptable. If the SAAM server detects any major discrepancy between the probe data and the LSA reports, then the server will try to verify the performance parameters with the next LSA reports. If the discrepancy still exists, the SAAM server will use the probing data over the LSA reports and generate an alert for the network administrator.

The probing approach will be first tested using PLAN (Packet Language for Active Networking). One or more PLAN packet(s) will be sent from the server to probe a particular set of links. The active packet(s) will collect performance information on the links and carry the data back to the SAAM server. The second implementation of server probing will be done using Java programming language, which will be integrated into the SAAM server/router prototype. The main reason for using PLAN for test purposes is to ensure that the technique/algorithm of server probing is feasible.

E. ORGANIZATION

In Chapter II, detailed background information is presented to explain the active networking approach and resident agent support for the SAAM prototype. Chapter III is devoted to the SAAM server probing approach and algorithms that perform this task. Chapter IV explains the server probing in Packet Language for Active Networking (PLAN) environment, and also explains the implementation of server probing in PLAN. Chapter V explains the server probing designed specifically for the SAAM prototype. Chapter VI is devoted to the server probing implementation and the observations. Finally, Chapter VII concludes the thesis with lessons learned and future work of this study.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

A. ACTIVE NETWORKING APPROACH

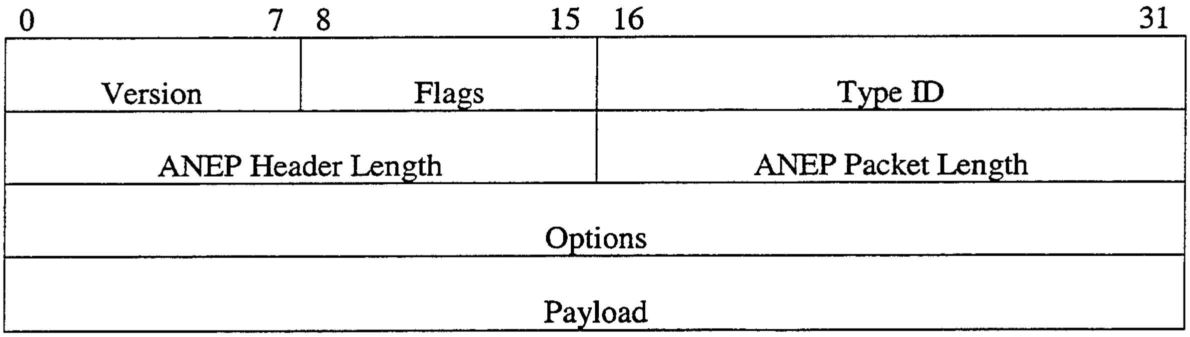
In the SAAM architecture, a SAAM Server will make routing and other important decisions on behalf of the routers in its region. In order to make the right decisions and support QoS (e.g., IntServ and DiffServ), the SAAM server needs to maintain an accurate region-wide view of network performance. This will be achieved as routers periodically send Link State Advertisement (LSA) messages to the SAAM server (Currently, the LSA messages report two key Link Performance Statistics, the average delay and the loss rate experienced by packets.) Moreover, the server needs to perform sanity checks of these statistics by probing specific links. The SAAM server will probe the links by dynamically injecting customized probing programs. In other words, the probing will be done with the active networking approach.

With the active networking approach, the network is no longer viewed as a passive mover of bits or bytes, but rather as a system capable of dynamically loading and executing programs written in a variety of languages. One can inject customized programs into the network, which can collect router state information, or even change some of the router's core modules (e.g., packet scheduler).

A typical active network consists of active nodes that have the ability to process active packets -- packets that carry a self-contained runnable code. The node's operating system is responsible for allocating and scheduling resources such bandwidth and the CPU. Each node also has one or more execution environments, each of which is a virtual machine that understands active packets written in a particular language. An example of such a virtual machine is the Java Virtual Machine (JVM). Another example is the Ocaml Virtual Machine. Execution environments will be explained in depth later in this chapter.

1. Active Network Encapsulation Protocol (ANEP)

As stated above, within the active networking approach the network is no longer passive; on the contrary, it is capable of dynamically loading and executing programs, written in different languages (e.g., PLAN, ANTS). The programs sent to an active node must be forwarded to their respective Execution Environment (EE). ANEP provides a mechanism for encapsulating active network packets for transmission over different media, and a proper de-multiplexing to the appropriate execution environment. Each Execution Environment has a type identifier number in the ANEP header, which is assigned by the Active Networks Assigned Numbers Authority (ANANA). The format of ANEP is shown in Figure 3.



0 7	8 15	16 31
Version	Flags	Type ID
ANEP Header Length		ANEP Packet Length
Options		
Payload		

Figure 3. Format of ANEP Header

The length of the *version* field is 8 bits, and indicates the header format in use. The active node will discard the packet if it does not recognize the packet's version number.

The length of the *flags* field is 8 bits, and indicates what the node should do if it does not recognize the *type ID*.

The length of the *type ID* field is 16 bits, and indicates where the packet will be evaluated. According to this field, the packet will be forwarded to the proper *execution environment*. As stated above, the authority for assigning *type ID* values is the ANANA.

The *ANEP header length* field is 16 bits, and specifies the length of the ANEP header in 32 bit words. The field value must be 2, if no options are included in the packet.

The *ANEP packet length* is 16 bits, and specifies the length of the entire packet including the packet payload. This field will serve to recover the packet if it has been transmitted over a lower layer that does not provide recovery of the packet length.

Options in the form of Type/Length/Value (TLV) can be included in the *options field.*

The active network packet format including the ANEP header is illustrated in Figure 4.

IP	UDP	ANEP	LOAD

Figure 4. Active Network Packet Format

2. Active Network Daemon (ANETD)

ANETD is an experimental daemon specifically designed to support the deployment, operation and control of active networks [Ref. 7]. It connects the active node to the Active Networking Backbone (ABONE), which is an experimental backbone to support active nodes with active packets.

ANETD performs two major functions.

- It allows the deployment, configuration and control of networking software (including current active networking execution environment prototypes) into the network. [Ref. 8]
- It de-multiplexes active network packets (encapsulated using ANEP) to multiple execution environments on the same network node and sharing the same input port. [Ref. 8]

The daemon listens to a user defined UDP port, and intercepts active packets encapsulated within an ANEP header. The daemon also checks the *type ID* field of the header, and de-multiplexes the active packet to the appropriate execution environment.

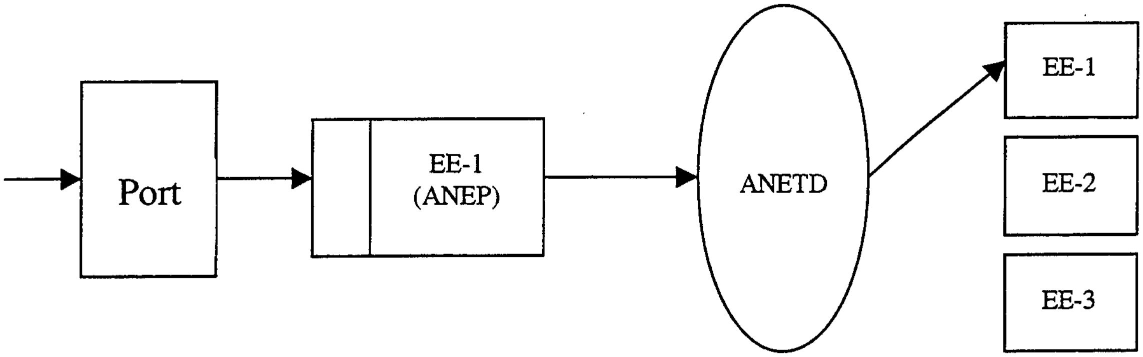


Figure 5. Illustration of De-multiplexing an Active Packet to EE-1

In Figure 5 we can see that an active packet arrives to a UDP port. Since ANETD will listen to that particular port, it will receive the packet, check for the *type ID* in the ANEP header, and will de-multiplex the packet to the appropriate execution environment. In this case it will de-multiplex the packet to EE-1.

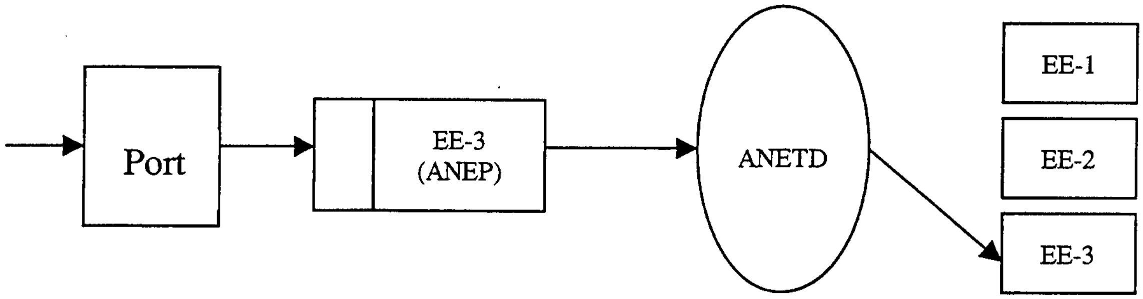


Figure 6. Illustration of De-multiplexing an Active Packet to EE-3

The same concept is shown in Figure 6. In this case the active packet is de-multiplexed to EE-3, because the type ID in the ANEP header shows that its appropriate execution environment is EE-3.

The deployment of networking software is similar to the concept of plug-in requirements enabling browsers to perform certain tasks (e.g., audio stream plug-in, video display plug-in). The network services are specified as Universal Resource Locator (URL). The daemon downloads the service with the Hypertext Transfer Protocol (HTTP) *get* command from the corresponding URL, and then strips off the Hypertext Markup Language (HTML) header from the received code and installs the service. This process is illustrated in Figure 7.

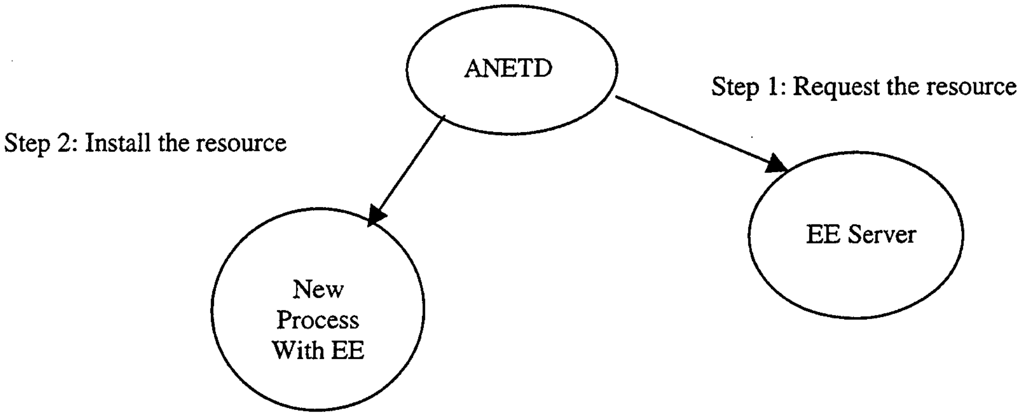


Figure 7. Illustration of the Deployment of an Active Network Resource

Currently, ANETD is available for Solaris on Sparc, Linux on x86 and FreeBsd on x86, and is implemented in C programming language. The daemon can be invoked as a user application and does not need any runtime privileges [Ref. 8]. The daemon will listen to port number 3322 by default, and the following command line on a computer with Linux Operating System (OS) will start the daemon..

ad.<operating system type>[-p <ANEP port>][-u<local port pool start>] [-s]

- *<operating system type>* the available options are
    - *solaris* = SunOS 5.5.x running on Sparc
    - *linux* = Linux running on Intel x86 platform
    - *bsd44* = FreeBsd running on Intel x86 platform
- *<ANEP port>* is the port number which ANETD will be listening to. The default value will be 3322.
- *<local port pool start>* is the starting port number for dynamically allocating local ports. The default value is 8000.

- *-s* authorizes ANETD to send heart-beat packets every 30 seconds to the main ANETD server, *sequoia.csl.sri.com.*

In order to illustrate the invocation of the ANETD, let's examine the following command line:

```
ad.linux –p 3326 –u 8800 –s
```

When this command line is executed, the ANETD will listen to UDP port 3326 for active packets. It will also use port 8800 for dynamically allocating local ports. At the same time, the daemon will send heart-beat packets every 30 seconds to the main ANETD server. From this command line, we can determine that the daemon is running on an Intel x86 processor based computer with Linux OS.

3. Execution Environments (EE)

As mentioned earlier in this chapter, an execution environment is used for evaluating the active packets. The daemon will de-multiplex the active packets according to the *type ID* field in the ANEP header, to its relevant EE. The *type ID* number that is associated with a particular EE is assigned by ANANA. Currently the ANANA is Bob Braden (braden@isi.edu) as of July, 1997.

There are couple execution environments available for download (e.g. ANTS, PLAN, Smart Packets). These execution environments are implemented by different institutions, and each claim advantages over the others. For research purposes, we have chosen Packet Language for Active Networking (PLAN) as the execution environment.

PLAN is implemented in both Java and Ocaml programming languages. To avoid confusion, the details of PLAN will be discussed in Chapter 4.

4. Active Networking Backbone (ABONE)

ABONE is an experimental backbone for active networking purposes. An interested party who wants connect to the backbone has to fill out the registration information available at http://www.csl.sri.com/ancors/abone/. This information will be available for other institutions experimenting with active networking. During the registration process, one will need to provide a public key. The key generation program is available from the same URL described above. Once you get the key generation program for your experimental platform, you can register and connect to ABONE. There are two important requirements once the node is connected to ABONE. First, the experimental node needs to be up and running. Second, the experimental node must be reachable through a non-standard UDP port, implying that one might need to apply to the network administrator for firewall configuration. After installing and invoking the ANETD, the node will be part of ABONE.

In order to implement SAAM server probing with the active networking approach, melon.cs.nps.navy.mil is registered and connected to ABONE. The ANETD version 1.0 runs under Linux OS. Recall that ANETD can run with Linux on Intel x86, FreeBsd on Intel x86 and Solaris on Sparc. The graphical status of ABONE is shown in Figure 8.

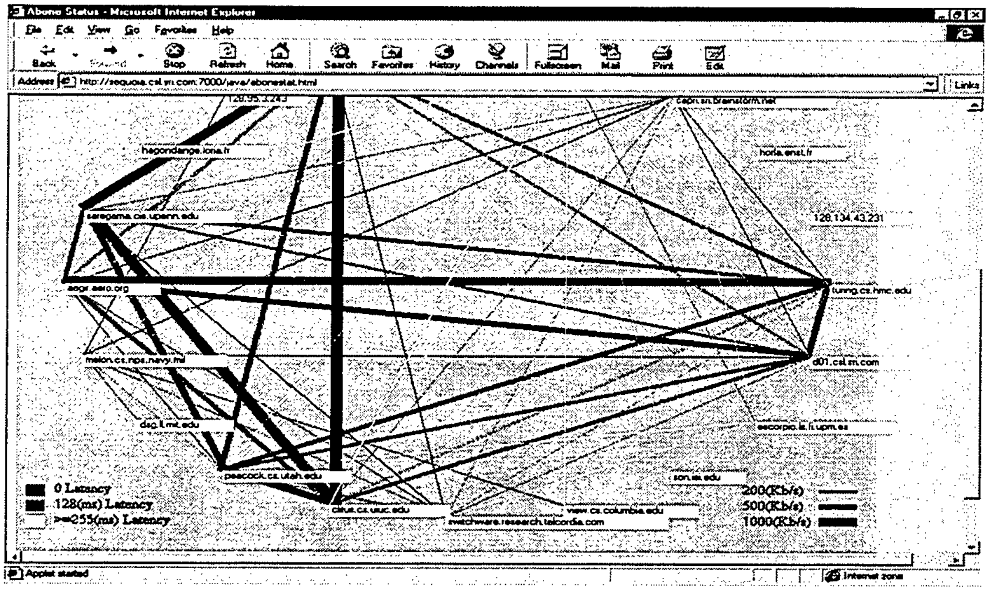


Figure 8. Graphical Status of ABONE

B. RESIDENT AGENT SUPPORT OF THE SAAM PROTOTYPE

A working prototype of the proposed SAAM architecture has been developed in the Java programming language. The main reasons for using Java as the programming language are its portability and support of resident agents, together which allow us to use the active network approach. It is desirable for the server to dynamically send some components, instead of leaving them to reside permanently on every router. This is a suitable approach to the lightweight router concept. The SAAM server simply sends the resident agents to the routers, which will be installed as a module and will perform its tasks. Once the task is completed the resident agent can be uninstalled, thus relieving the router from performing extra tasks or having extra components. The resident agent support also allows us to update some modules on the router.

The following quote from *Programming and Deploying Java Mobile Agents with Aglets*^{TM}, Danny B. Lance/Mitsuru Oshima, page 35, summarizes the concept of resident agents:

"A key feature of mobile agents is that they can be serialized and deserialized. Java conveniently provides a built-in serialization mechanism that can represent the state of an object in a serialized form sufficiently detailed for the object to be reconstructed later. The serialized form of the object must be able to identify the Java class from which the object's state was saved and to restore the state in a new instance. Objects often refer to other objects. To maintain the object structure, these other objects must be stored and retrieved at the same time. When an object is stored, all the objects in the graph that are reachable from the object are also stored."

III. SAAM SERVER PROBING APPROACH AND ALGORITHMS

A. LINK STATE ADVERTISMENT (LSA) MESSAGES

In the SAAM architecture, a SAAM server will make routing and other important decisions on behalf of the routers in its region. In order to make the right decisions, the SAAM server always needs to maintain an accurate region-wide view of network performance. This accurate view will be achieved as routers periodically send LSA messages to the server.

Currently, the LSA messages report two key Link Performance Statistics (LPS), the average delay and the loss rate experienced by packets. These LPSs are calculated for every service level pipe of each router's interface. The LSA messages will serve as a base for the server to develop an accurate picture of regional paths crossing multiple service level pipes for QoS. The concept of service level pipes is illustrated in Figure 9.

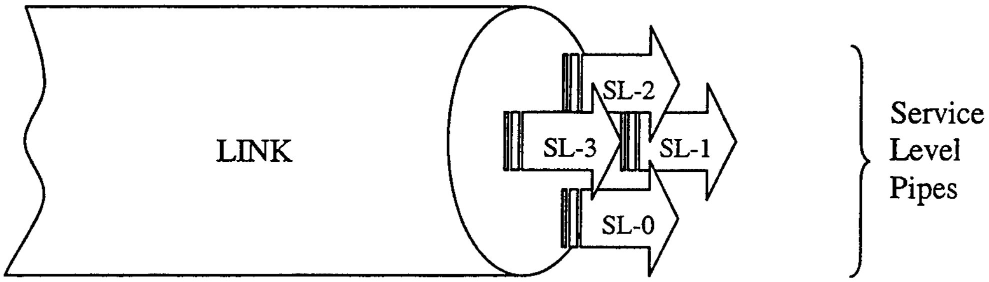


Figure 9. Service Level Pipes in a Link

Each service level pipe may have multiple input queues. Four separate service levels are assigned respectively to *control traffic*, *guaranteed service traffic*, *differentiated service traffic*, and *best effort traffic*.

B. SERVER PROBING FOR SANITY CHECK

The SAAM server will build the region-wide picture of network performance according to the LSA messages. For verification purposes, the server needs to perform a sanity check of these statistics by probing specific links.

Another important reason for verifying the accuracy of the statistics sent with the LSA messages is related to SAAM security. A compromised node can feign a good performance, and might mislead the SAAM server in making decision. This will certainly affect the QoS of many flows using that link.

There are two potential explanations as to why a node feigns good performance parameters. The first is that the router might be misconfigured and does not perform its tasks correctly. The second is that destructive individuals tamper with the traffic flow of a SAAM region, and degrade the overall network performance.

Since the SAAM server will make decisions according to LSA statistics, it is important that the server has an independent method to verify the validity of LSA data.

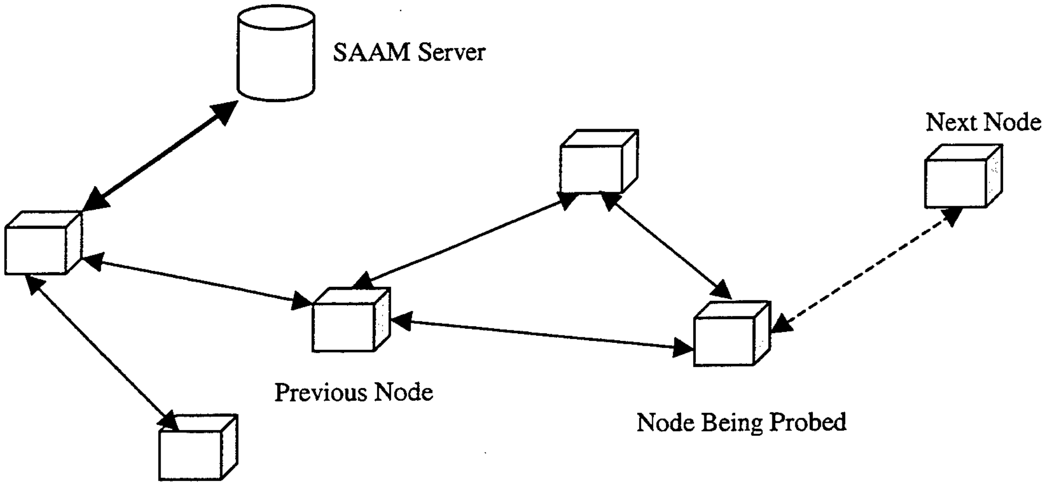


Figure 10. SAAM Server Probing

Let's first explain the terminology used in Figure 10. The node we wish to probe will be referred to as, *node being probed.* The previous neighbor of the probed node will be referred to as, *previous node*. The next neighbor of the probed node will be referred to as, n*ext node*.

The *previous node* will receive a command from the server to generate probing packets for the particular flow of a service level. The *next node* will also receive a command from the server to collect the probing results of a particular flow. After the *next node* collects the probing parameters, it will send the results back to the server. Then, the server will compare the parameters with those sent with the LSA messages. If the difference between the LSA result and the probe result is not within acceptable limits, the server will try to verify the information. If the discrepancy still exists after the first attempt to verify, the SAAM server will use the probing data over the LSA reports and generate an alert for the network administrator.

C. BANDWIDTH MEASUREMENT TECHNIQUES

Current bandwidth measurement techniques have many problems [Ref. 9]:

- Poor accuracy;
- Poor scalability;
- Lack of statistical robustness;
- Poor agility in adapting to bandwidth changes;
- Lack of flexibility in deployment;
- Inaccuracy when used on a variety of traffic types.

Developed by Stanford University, the Packet Pair Algorithm is one of the techniques used to measure the bottleneck bandwidth of a route with reasonable accuracy.

The bottleneck bandwidth of a route is the ideal bandwidth of the lowest bandwidth link (the bottleneck link) on that route between two hosts. In most networks, as long as the route between two hosts remains the same, the bottleneck bandwidth remains the same.

The bottleneck bandwidth is not affected by other network traffic. The available bandwidth of a route is the maximum bandwidth at which a host can transmit at a given point in time along that route. Available bandwidth is limited by other traffic along that route. [Ref. 9]

One of the most popular and simple techniques is to use throughput as an approximation of bandwidth. Throughput is the amount of data transported per unit of time. One drawback to this approach is that packet drop rate may have a significant effect on throughput, although not affecting bandwidth.

TCP uses a technique where it sends more and more packets until one is dropped. TCP estimates the bandwidth to be somewhere between the sending rate when the packet was dropped and half that rate. This approach has several flaws: [Ref. 9]

- TCP is measuring the bottleneck router's buffer size in addition to the bottleneck bandwidth;
- TCP wastes network resources by forcing a dropped packet and filling the router's buffers;

- TCP has to increase its sending rate slowly, or else it will over-shoot the real bandwidth and cause massive packet loss.

There are two algorithms to measure the bottleneck bandwidth. The first is the *Pathchar Algorithm,* and the second is the *Packet Pair Algorithm.* We will not discuss the Pathchar Algorithm in further detail, as one of its drawbacks is that its heavy consumption of network bandwidth resources. If this algorithm were used for a large number of hosts, it would consume too much bandwidth to be used regularly. As an approximation, for a 1-hop Ethernet network with a latency of 1ms, the average bandwidth consumed would be around 6 Mb/s.

D. PACKET PAIR ALGORITHM

For SAAM server probing, the performance parameters sent with the LSA messages should be verified. In order to measure the throughput of a particular link, we want the server sending a notification message or a resident agent to the *previous node*, so the *previous node* can initiate the probing process. The throughput measurement will be done using the *Packet Pair Algorithm.*

The principle of *Packet Pair Algorithm* is quite simple but very practical. Once the *Previous Node* receives a command or a resident agent, it will generate two packets for a particular flow of a service level with a signature to identify that the incoming packet is a probing packet. When the two packets are queued next to each other at the link, then they will exit the link *t* seconds apart. Since we know the packet size, we can simply calculate the throughput of the link with the following equation:

$$bandwidth(bnl) = \frac{size}{time(t)}$$

Equation 1. Mathematical Equation of Packet Pair Algorithm

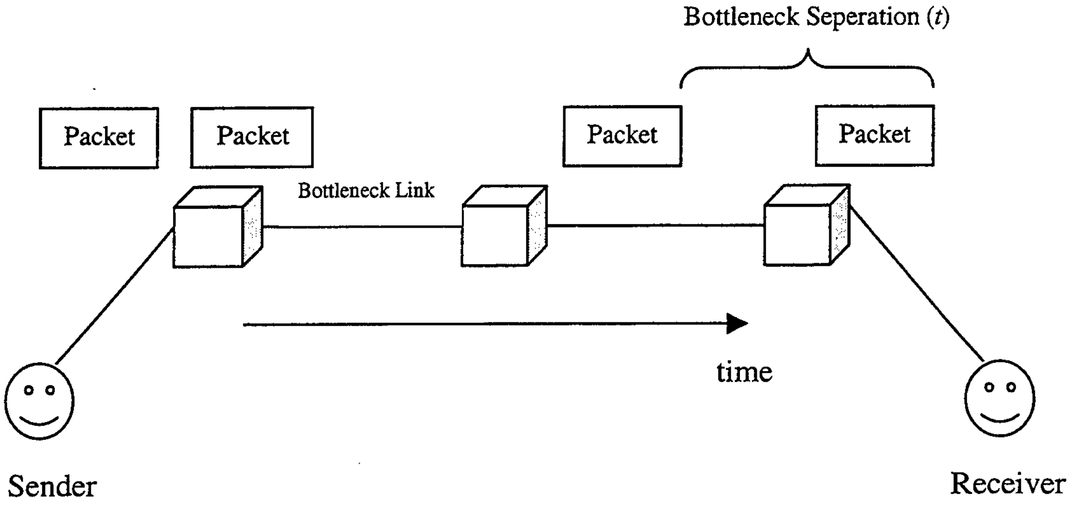


Figure 11. Packet Pair Algorithm

The variable *size* in the equation indicates the second packet's size. However, in our approach, the *size* variable is irrelevant, as the *previous node* will always send two completely identical packets. There will a separation between the two packets after they will leave the bottleneck link, which is the *bottleneck separation.* Since there are no links with lower bandwidth than the bottleneck link downstream from that link. Assuming the packets are the same size, the second packet will never catch up to the first packet. Within our approach the *next node*, which is the neighbor node of the n*ode being probed* will record the time of the bottleneck separation with the packet size, and calculate the bandwidth for the link. The *Next Node* will distinguish the probing packets from the regular data packets with the signature placed in the payload.

The two packets must be the same size, because different size packets have different velocities. If the second packet were smaller than the first, then its transmission delay would always be less than the first packet's. Consequently, it would pass through links faster than the first packet, that of quickly eliminating the *bottleneck separation.* Similarly, if the first packet is smaller, then it will be faster than the second packet, and continuously lengthen the bottleneck separation [Ref. 9].

THIS PAGE INTENTIONALLY LEFT BLANK

IV. IMPLEMENTATION OF SERVER PROBING IN PLAN

PLAN is an active network programming language developed by the SwitchWare project at University of Pennsylvania [Ref 10]. Based on calls to node-resident service routines, PLAN provides a restricted set of primitives and data types. PLAN programs are meant to execute remotely, a feature that makes it difficult to determine the cause of an unexpected PLAN program behavior. Therefore, it is important to provide apriori assurances about a program's behavior. The strong type check feature of PLAN makes debugging easier. Another PLAN property aimed at protecting network availability is the guarantee that PLAN programs use a bounded amount of network resources.

A. PLAN DISTRIBUTION

Currently, there are two implementations of the PLAN programming environment. The first implementation is written in Java programming language, and the version is 2.21 at the time of the publication of this thesis. The second implementation is written in the Ocaml programming language, and its current version is 3.2. The following subsections will explain both implementations.

1. Java 2.21 Distribution of PLAN

Java is a programming language developed by Sun Microsystems. The main advantage of Java is its portability, a characteristic that enables users to run Java programs on a great variety of operating systems. The compilation of the Java source code generates byte codes, which will be interpreted by the Java Virtual Machine (JVM).

The distribution of the Java version of PLAN includes class files, source code, documentation and sample programs. You can get the Java class files and execute them directly or you can get the source code and build it yourself. The file can be downloaded from **http://www.cis.upenn.edu/~swithware/PLAN/download.html**. Since we were using RedHat Linux 5.1 for our active networking test bed, we downloaded the file for the Unix OS.

Three packages are needed to install the Java distribution of PLAN.

(1) Java Development Kit (JDK) 1.1.x. The download and installation procedures are explained in APPENDIX A.

(2) Pizza version 0.39. This package is a substantial companion to JDK. The download and installation procedures are explained in APPENDIX B.

(3) The Java Compiler Compiler (JavaCC) version 0.6.1. The download and installation procedures are explained in APPENDIX C.

After these three packages are installed, one can unpack the source code of PLAN under a directory of his desire. One needs to compile the source code to generate the class files, or one can simply download the class files and install them.

2. Ocaml 3.2 Distribution of PLAN

Caml is a programming language developed by INRIA (French Research Institute for Computer Science) and is freely available. Ocaml (Objective Caml) is a version of the

Caml programming language. Ocaml is a modular system with support to the object-oriented paradigm, and includes an optimizing compiler.

One of the main objectives of the language is safe programming. That's why the compiler performs many sanity checks on source code before compilation. Ocaml programs are statically type-checked, a process that reduces programming errors. In addition to these features, Ocaml is a functional programming language.

The most recent version of the PLAN distribution written in Ocaml is 3.2 . The PLAN implementation is changed from Java to Ocaml programming language to improve the overall performance and safety of PLAN programs. The Ocaml version 3.2 is built with the following available packages:

- Ocaml version 2.02
- CamlP4 version 2.02, which is a preprocessor of Caml.
- Ocaml patch to enable Ethernet access on computers running Linux OS.

Download and installation procedures are explained in APPENDIX D.

B. PLAN ENVIRONMENT AND PROGRAMMING

1. PLAN Packets

In the active networking approach, the network is no longer viewed as a passive mover of packets, but rather as a capable of dynamically loading and executing programs. PLAN programs are bundled into packets, injected into a network, and then evaluated.

After the evaluation, the packet might create other packets, which may then be evaluated at other nodes in the network.

A PLAN program consists of two components, the *code* and the *bindings*. The bindings serve as resource constraints, so all PLAN programs are guaranteed to terminate and consume limited amount network resources [Ref.10].

2. Model of PLAN Evaluation

In this section, the PLAN evaluation environment is explained to promote a better understanding of how PLAN programs are evaluated in an active networking environment.

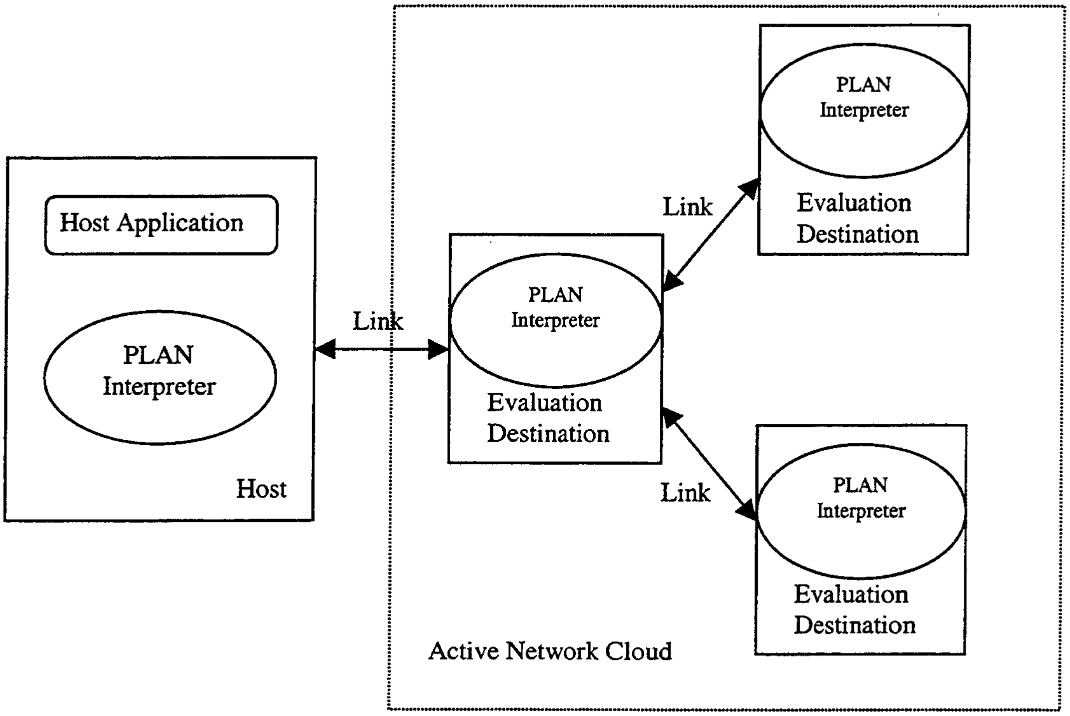


Figure 12. The PLAN Evaluation Environment

Consider Figure 12. A PLAN network consists of hosts, which form the endpoints of the environment, and routers, which form the cloud. We will refer hosts and routers as *nodes* throughout the rest of this chapter. Hosts are distinguished from routers in that they may serve host applications, forming part of a PLAN application [Ref.11]. All the nodes in the PLAN evaluation environment are capable of evaluating PLAN programs.

When sending a PLAN packet to the active network cloud, one must send it from the host node, where a PLAN packet begins its life cycle. The packet will be injected into the active cloud after being passed to the PLAN interpreter through a port. (The results from the active nodes executing the PLAN packet will be received through this same port to be displayed as standard output.) The local interpreter then proceeds to route the packet to its evaluation destination, which is one of the nodes in the active cloud. At each "hop" on the way to its evaluation destination, the resource bound of the packet is decremented to limit the use of network resources. In case packet's resource bound is exhausted, the packet is terminated, and a message is sent back to the source for potential recovery of the data [Ref. 11].

The PLAN packet is evaluated when the packet has reached its evaluation destination. First, the code is parsed top-down to register the top-level bindings. All function bindings respect static scooping; that is, the set of bindings (available to a function when it is evaluated) are those that were available when the function was defined. Evaluation then begins with the function call defined by the invocation. Any value that would have been returned by the function is discarded. [Ref. 11]

3. Starting PLAN Aware Routers

One can set up more than one PLAN-aware routers on a computer. These routers will have the ability to process and forward PLAN packets. ANETD is not essential for the PLAN environment for de-multiplexing PLAN packets. PLAND (PLAN Daemon) is a replacement for ANETD, and performs the same function. It de-multiplexes PLAN packets to the PLAN execution environment, and serves as the entry point of the active node for a PLAN packet. The PLAN distribution has the following executable files [Ref. 12].

- *pland*: The PLAN daemon implementing an Active Router.
- *inject*: Sends a PLAN program to the local *pland* to be interpreted.
- *plan*: A non-network PLAN interpreter in *read-eval-print* mode used to test out basic programs.

In order to set up an active router on a computer, one needs to invoke the PLAN daemon by typing the following command. (The *-help* switch will give information about the usage of this command.)

```
% bin/pland –help
```

The following help message will be displayed as the output of this command:

```
usage: bin/pland [-router] [-firewall d1, d2, ...dn] [-l log] [-ip port] [-rf rout_tab_files]
        [-policy policy_file] [-authlist h1, h2, ...hn] ifc_spec_file
        -router : Turn on router mode.
        -firewall: Specify devices to firewall (default is none)
        -ip: Specify static route table file.
```

-hf: Specify host name file (default=*./key.<local portnum>*).

-policy: Specify policy specification file (depends on policy engine in use).

-authlist: Specify nodes to authenticate with (default is none).

Let's build an active cloud with two active routers to illustrate a step by step procedure. First of all we have to define the interface files. Let's call them respectively m1, m2, and the contents of those files will be as follows:

File m1 contains:

```
1 ip0 ip melon.cs.nps.navy.mil:3324 melon.cs.nps.navy.mil:3325
```

File m2 contains:

```
1 ip0 ip melon.cs.nps.navy.mil:3325 melon.cs.nps.navy.mil:3324
```

Remember, we are creating those active nodes on the same computer, so it is a virtual active network cloud with two nodes. The first file indicates that the node *m1* has one IP interface named *ip0* with the address, "melon.cs.nps.navy.mil:3324", and its neighbor has the interface address, "melon.cs.nps.navy.mil:3325". The file *m2* has also one IP interface named *ip0.*

Before we can start running our active nodes, we must add the IP number and the identification name of our local computer to the *EXP_IP_ADDRS* file. Once accomplished, we can run the active routers on our local computer by invoking the PLAN daemon with the appropriate flags:

```
% bin/pland –l log3324 –ip 3324 m1

% bin/pland –l log3325 –ip 3325 m2
```

This command sequence will call the *m1* and *m2* files and create the active routers. If we don't specify any port number, the default number for the PLAN daemon is 3324.

4. Injecting PLAN Packets To Active Cloud

Thus far, we have built an active cloud, which is ready to receive PLAN packets. Let's create a simple PLAN program called HelloWorld.plan, and then inject it to the active network to illustrate a step by step procedure. The program contains the following code:

```
svc print : ‘a -> unit
fun start() =
        ( print( “This is a hello message for first programming experience \n”) )
```

This program has just one function named *start()* with no parameters, and will simply display the message string as output. In order to inject this packet we need to use the command below. The *–help* flag shows us its usage.

```
    % bin/inject –help
usage: bin/inject [-v] [-val] [-p port] [-ed evalDest] [-rf routFun] [-hf host_file]
    [-o outfile] <codefile> <Resource Bound>
    -v: Turn on verbose mode
    -val: Unmarshall and print responses as values (rather than strings).
```

-p: Set initial evaluation destination.

-rf: Specify routing function.

-fid: Specify flow ID (default is 0).

-hf: Specify file for hostname resolution (default is the *EXP_IP_ADDRS* file).

-o: Specify file to output marshalled packet (default is none).

A sample of a command to inject the "HelloWorld.plan" program would be as follows:

```
% bin/inject HelloWold.plan 10
```

Since we didn't specify the port in which to inject the packet, the default port number will be used by the daemon. We also specified the resource bound *take 10.* After injecting the program, the PLAN code will wait to receive the next input, which is the *initial invocation.* Recall that a PLAN program is a list of definitions, and when the program arrives at an active host, one must specify which function to start executing, and with what arguments [Ref.11]. In the "HelloWorld.plan" case, the function *start( )* is not expecting any parameters, so we simply type at the command line:

```
start();
```

After the invocation of the function, we will receive the output message string of the program.

```
% This is a hello message for first programming experience
```

This sample program illustrates the simple procedures to build an active network cloud, consisting of active nodes, and injecting active PLAN packets to this network. The following section will explain the implementation of the *Packet Pair Algorithm* in PLAN.

C. IMPLEMENTATION OF PACKET PAIR ALGORITHM IN PLAN

It is important for the SAAM server to have the most accurate performance parameters of the links, to build the maintain an accurate region-wide view of network performance. These performance parameters will be sent by the nodes using LSA messages. Moreover, the server needs to perform sanity checks on these statistics by probing specific links. The probing should not be detected by the *node being probed* at the same time for security considerations mentioned in Chapter III. For throughput probing of a specific link, the server will use the Packet Pair Algorithm. The implementation of this algorithm in PLAN is presented in APPENDIX E.

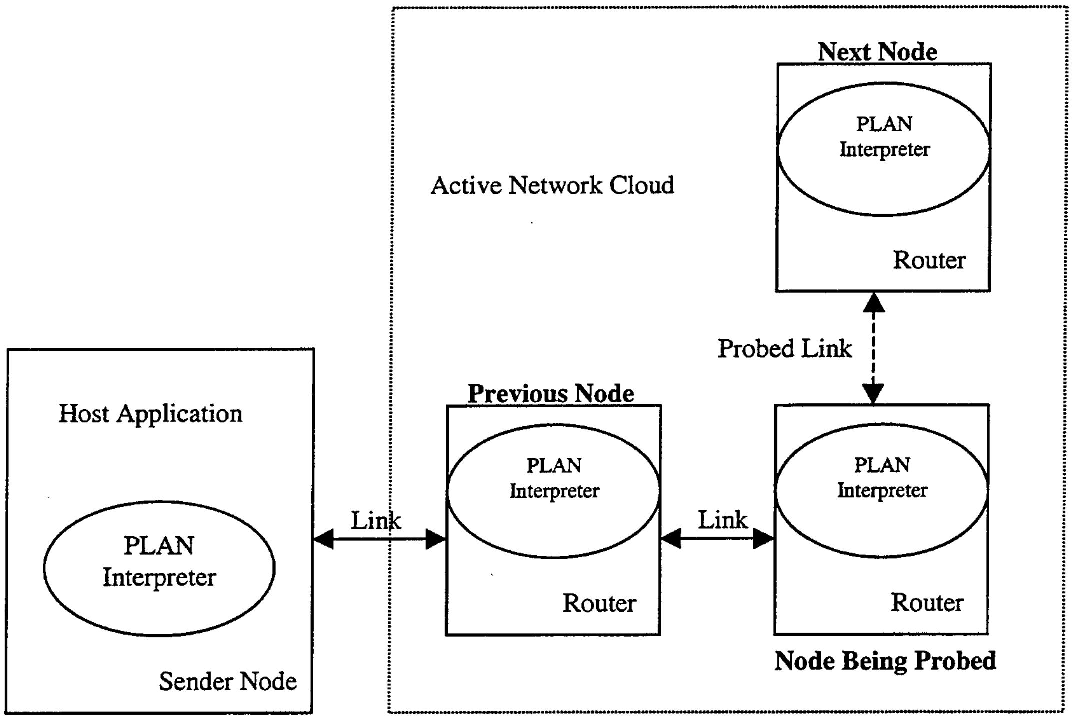


Figure 13. Packet Pair Probing with PLAN

In Figure 13, a topology with three active nodes is shown, which allows injecting the *packet pair* program. First, the *packet pair* PLAN packet is injected into the *previous node*. When the packet reaches its evaluation destination, or the *previous node*, the PLAN packet will generate two other packets with the address information of the host node, providing a return of the acknowledgment packets, generated by the *OnRemote()* service call. The two packets are generated with the following code snipped.

```
if(thisHost() = previousNode ) then
        OnRemote(|sendpacket|( senderNode), nextNode, 50, defaultRoute)
else();
if(thisHost() = previousNode ) then
        OnRemote(|sendpacket|( senderNode), nextNode, 50, defaultRoute)
else()
```

The syntax for *OnRemote()* is as follows:

OnRemote(E, H, Rb, Routing)

The meaning of *OnRemote()* service call is, "evaluate E on node H, and use the routing function to determine how to get to node H." The resource bound restriction is passed with the variable *Rb*.

The sendpacket( ) method that creates two acknowledgement packets including the time stamp values is as follows:

```
fun sendpacket( senderNode:host )=
(
        let val record_time1:int*int = gettimeofday( ) in
        OnRemote( |ack|(thisHost(), record_time1), senderNode, 30,
                defaultRoute)
        end
)
```

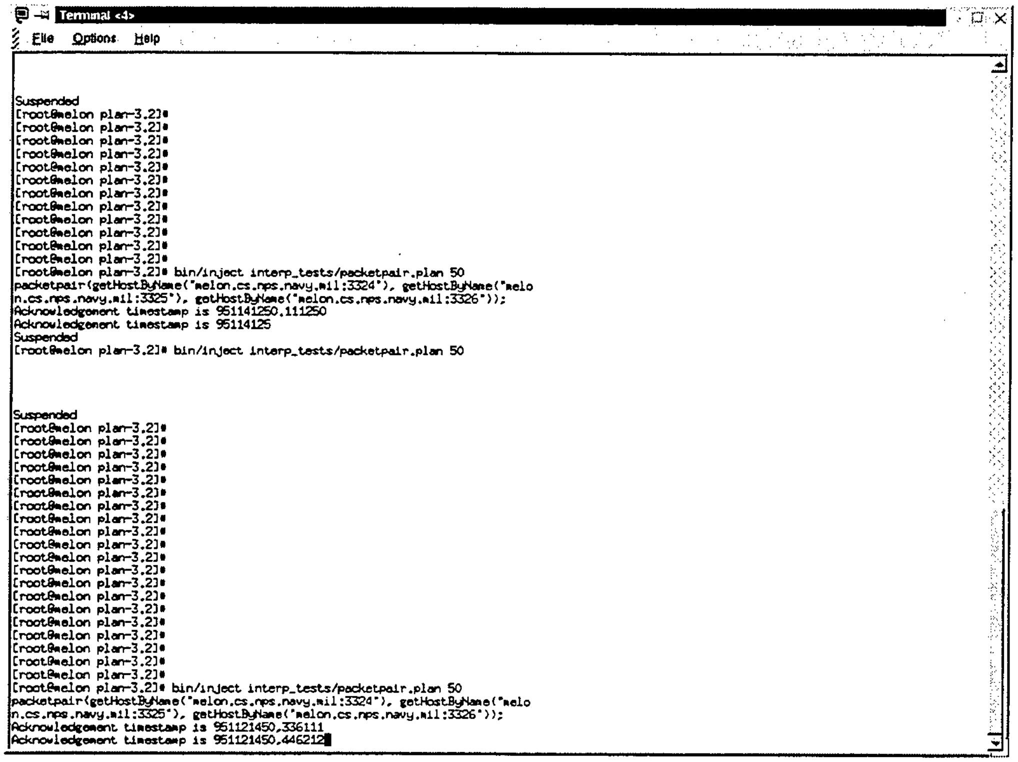
When the two created packets arrive at the *next node*, each of them will send one acknowledgment packet back to the *sender node,* including a time stamp showing its arrival at the *next node*. When the acknowledgment arrives at the *sender node,* (this is the equivalent of the SAAM server node) the *sender node* will simply display the identification of the *next node* with the time stamp.

```
fun ack( where:host, time:int*int) =
( print( “Host is :”); print( where );print( “Time stamp is “);print(time) )
```

With the time stamp information, the *sender node* can calculate the packet separation of the two acknowledgment packets. Dividing the packet size of the acknowledgements by the time separation will give us the throughput metric of the probed link, which is the dotted link in Figure 13. The test results for getting the time stamps of the acknowledgement packets are shown in Table 1, and a snapshot of the command window displaying the time stamp values are shown in Figure 14.

Packet	Timestamp (sec)	Throughput (Kbps)
Acknowledgement 1	9511215860, 336145	49
Acknowledgement 2	9511215860, 558178	
Acknowledgement 1	9511215868, 189663	20
Acknowledgement 2	9511215868, 777145	
Acknowledgement 1	9511215875, 568896	30
Acknowledgement 2	9511215875, 963346	

Table 1. Packet Pair Test Results



```
Suspended
[root@melon plan-3.2]#
[root@melon plan-3.2]#
[root@melon plan-3.2]#
[root@melon plan-3.2]#
[root@melon plan-3.2]#
[root@melon plan-3.2]#
[root@melon plan-3.2]#
[root@melon plan-3.2]#
[root@melon plan-3.2]#
[root@melon plan-3.2]#
[root@melon plan-3.2]#
[root@melon plan-3.2]#
[root@melon plan-3.2]# bin/inject interp_tests/packetpair.plan 50
packetpair(getHostByName("melon.cs.nps.navy.mil:3324"), getHostByName("melo
n.cs.nps.navy.mil:3325"), getHostByName("melon.cs.nps.navy.mil:3326"));
Acknowledgement timestamp is 951141250.111250
Acknowledgement timestamp is 95114125
Suspended
[root@melon plan-3.2]# bin/inject interp_tests/packetpair.plan 50



Suspended
[root@melon plan-3.2]#
[root@melon plan-3.2]#
[root@melon plan-3.2]#
[root@melon plan-3.2]#
[root@melon plan-3.2]#
[root@melon plan-3.2]#
[root@melon plan-3.2]#
[root@melon plan-3.2]#
[root@melon plan-3.2]#
[root@melon plan-3.2]#
[root@melon plan-3.2]#
[root@melon plan-3.2]#
[root@melon plan-3.2]#
[root@melon plan-3.2]#
[root@melon plan-3.2]#
[root@melon plan-3.2]#
[root@melon plan-3.2]#
[root@melon plan-3.2]#
[root@melon plan-3.2]#
[root@melon plan-3.2]#
[root@melon plan-3.2]# bin/inject interp_tests/packetpair.plan 50
packetpair(getHostByName("melon.cs.nps.navy.mil:3324"), getHostByName("melo
n.cs.nps.navy.mil:3325"), getHostByName("melon.cs.nps.navy.mil:3326"));
Acknowledgement timestamp is 951121450.336111
Acknowledgement timestamp is 951121450.446212
```

Figure 14. Snapshot of the Command Window on "melon.cs.nps.navy.mil"

THIS PAGE INTENTIONALLY LEFT BLANK

V. SERVER PROBING DESIGN FOR THE SAAM PROTOTYPE

A. SAAM PROTOTYPE

1. Overview

A working prototype of the proposed SAAM architecture was developed by two former Naval Postgraduate School students, Dean Vrable, and John Yarger, using the Java programming language. The main reason for using Java as the programming language is its portability. Java programs can run on any platform without changing the precompiled byte code.

IPv6 supports flow routing. In order to take advantage of this support, an emulated environment was built to route IPv6 packets at the application layer over an IPv4 network.

The prototype consists of a router module and a server module. The router is designed to maintain a flow routing table as well as an Address Resolution Protocol (ARP) cache table. The router manages its functionality by using the standard concept of transport layer ports.

The server has two primary tasks. The first task is to maintain an accurate status of its region. This task is accomplished through the building of a Path Information Base (PIB) from auto configuration messages and Link State Advertisement (LSA) messages. The second task is to respond to flow requests. This task is accomplished by accessing the PIB in order to identify the optimal path for a requested flow from the set of paths that can support the request. If this optimal path can be found, then the routers in the path are sent appropriate update for their flow routing tables. Finally, the requesting application is notified of the assigned *flow ID*, if the flow can be supported. [Ref. 3]

2. Resident Agent Support

The probing component should be sent dynamically by the server, instead of residing permanently on every router. Java is perfectly suited for this approach. The SAAM prototype is designed to support resident agents, allowing the components of a router to be upgraded and installed dynamically during runtime. The precompiled byte code of a resident agent is registered as a new module of the receiving node. The module may run itself by creating a thread, or it may be called by an existing thread. For instance, after a node in the prototype receives a server resident agent it becomes a SAAM server, otherwise it stands up as a router.

The following quote from *Programming and Deploying Java Mobile Agents with Aglets*^{TM}, Danny B. Lance/Mitsuru Oshima, page 35, summarizes the concept of resident agents:

"A key feature of mobile agents is that they can be serialized and deserialized. Java conveniently provides a built-in serialization mechanism that can represent the state of an object in a serialized form sufficiently detailed for the object to be reconstructed later. The serialized form of the object must be able to identify the Java class from which the object's state was saved and to restore the state in a new instance. Objects often refer to other objects. To maintain the object structure, these other objects must be stored and retrieved at the same time. When an object is stored, all the objects in the graph that are reachable from the object are also stored."

B. SERVER PROBING APPROACH AND ANALYSIS

The server needs to install the probing components dynamically to the routers during runtime. In order to probe a link we have the concept of a *previous node*, and a *next node*.

The server sends a resident agent to the *previous node* to initiate the probing process. Before the probing starts, the resident agent needs to obtain certain parameters from the server to create the appropriate probing packets. This will be achieved as the server sends a *probing activation message* to the *previous node*. The server sends another resident agent to the *next node* to intercept the probing packets and compute the throughput metric. The server will also send the required parameters to the *next node probing resident agent* with an activation message. Sending activation messages allows a generic resident agent which is capable of performing various server probing (i.e. throughput, loss rate, latency, utilization). The activation message determines what kind of task to perform. (The format and parameters of the activation messages will be explained later in this chapter.)

After the *next node resident agent* receives the *activation message* the resident agent will monitor the packets and will look for the probing packets to compute the performance metrics by identifying an embedded signature in the last four bytes of the payload.

In the activation message, the server sends a four byte *probe ID,* which is an arbitrary number to keep track of probing results for various links. This *probe ID* is also used as the signature in the payload. The ordinary and the probing packet formats are shown in Figures 15 and 16 respectively.

40 bytes	Variable
IPv6 Header	Payload

Figure 15. Data Packet Format

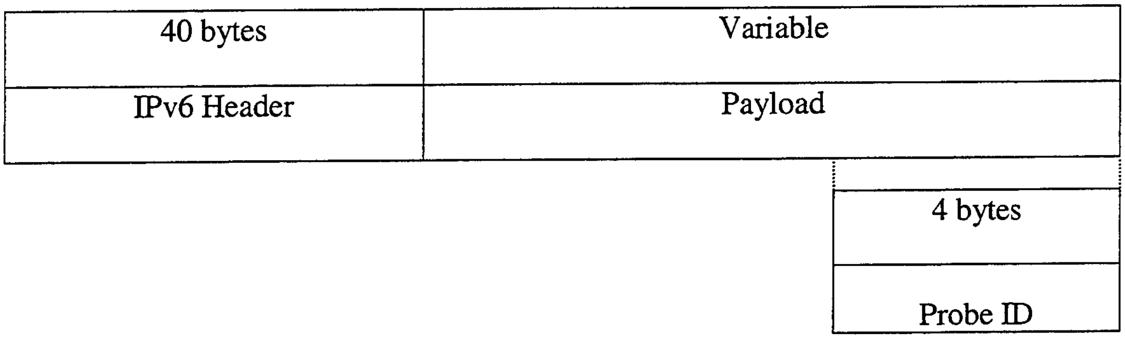


Figure 16. Probing Packet Format

1. Previous Node Activation Message Format

After the server sends the resident agent to the *previous node*, the server needs to send to the agent certain parameters to customize the probing process (e.g., IPv6 header to use for probe packets, probe ID, etc.).

The server first determines a particular flow to mask probe packets, and then sends a copy of IPv6 header for that particular flow to the probe agent. Recall that the server is the central authority, and has information about flows in its region. The *previous node activation message* has four parameter fields, shown in Figure 17.

The *type of probing* field will identify the kind of probing the *previous node resident agent* will initiate. For instance, if we want to measure the throughput metric, the resident agent will use the Packet Pair Algorithm as the probing process. Or, if we want to measure the loss rate metric, then the resident agent will initiate a different probing process (i.e., sending a set of packets).

The probe identification will be used as the signature for the probing packets, to differentiate probe packets from the ordinary data packets.

Once the probing process is completed and the agent instantiates the probe result message, the resident agent uses this probe identification with the associated probe result. This system provides a way of keeping track of probe results on the server side.

The *payload length* will determine the size of the Ipv6 probing packets.

1 byte	1 byte	4 bytes	40 bytes	2 bytes
Type ID	Type of Probing	Probe ID	IPv6 Header	Payload Length

Figure 17. Previous Node Activation Message Format

Type ID: Is 1 byte in length and is a predefined identification number. All Previous Node Activation (PNA) messages will have a *type ID* of 13.

Type of Probing:

0	Delay (ms)
1	Loss Rate (0.01%)
2	Throughput (Kbps)
3	Utilization(0.01%)

Probe ID: This probe identification number will be used as the signature in the payload of the probing packets.

IPv6 Header: This header information is a copy of the IPv6 header of a particular flow the server wants to probe.

Payload Length: Will determine the size of the probing packets.

2. Next Node Activation Message

After the *next node resident agent* is installed, it requires certain parameters to initiate the measurement process (e.g., probe ID, source address).

The format of the *next node activation message* is shown in Figure 18.

The *type ID* will identify that this message is a next node activation message. This is a predefined identifier, and it has a value of 14.

As it was for the previous node activation message, the *type of probing* field will determine what kind of probing is in progress, and what metric the next node resident agent shall compute.

The *probe ID* will be used to check for the signature in the data packets. The probing packets are the same as the ordinary data packets, except for the signature in the end of its payload.

With the *IPv6 header* information, the resident agent will determine what kind of packets to monitor, and the *measurement interval* field will determine how long the resident agent will remain active for packet monitoring. After the *measurement interval* expires, either the resident agent will be uninstalled, or will remain passive waiting for another activation message to perform another measurement.

The *NIC to be monitored* field will tell the resident agent which network interface to monitor for probe packets.

1 byte	1 byte	4 bytes	40 bytes	2 bytes	1 byte
Type ID	Type of Probing	Probe ID	IPv6 Header	Measurement Interval (ms)	NIC to be Monitored

Figure 18. Next Node Activation Message Format

Type ID: Is 1 byte in length and is a predefined identification number. All Next Node Activation (NNA) messages will have a *type ID* of 14.

Type of Probing:

0	Delay (ms)
1	Loss Rate (0.01%)
2	Throughput (Kbps)
3	Utilization(0.01%)

Probe ID: This probe identification will be used to detect the signature in the last four bytes of the payload in the probing packets.

IPv6 Header: This is a copy of the IPv6 header of a particular flow the server wants to probe. The resident agent will determine what type of traffic it should monitor.

Measurement Interval: This value will determine for how long the resident agent will monitor probing packets.

NIC to be Monitored: This value will identify the network interface card to be monitored for incoming probing packets.

3. Probe Result Message

Once the probing process has ended and the *next node probing agent* has performed the metric performance computation, the agent sends back the results to the server. The results will be encapsulated in a message format shown in Figure 19.

1 byte	4 bytes	1 byte	3 bytes * Number of Results	
Type ID	Probe ID	Number of Results	Type of Result (1byte)	Measurement Result (2bytes)

Figure 19. Probe Result Message Format

Type ID: It is 1 byte in length and is a predefined identification number. All Probe Result (PR) messages will have a *type ID* of 15.

Probe ID: The server will identify for which link this probing result is for, and will keep track of all the ongoing probes. It will assign a unique *probe ID* for each probing process.

Number of Results: The message format may contain more than one probing result. The next node resident agent may send all four types probing results in a single message.

Type of Result:

0	Delay (ms)
1	Loss Rate (0.01%)
2	Throughput (Kbps)
3	Utilization(0.01%)

Measurement Result: The probing result for the particular *type of probing*, and *probe ID*.

4. Flow Chart And Timing Diagram of Server Probing Process

The server initiates a probing process by sending a resident agent to the *next node*, which is capable of collecting and computing probing statistics. This resident agent will wait for an activation message, that will tell the resident agent which network interface to monitor, how long the monitoring task must be performed, and most importantly, the signature *(probe ID)* of the probing packets.

The server then sends another resident agent to the previous node, which is capable of constructing probing packets and passing them to an output interface as regular packets. This resident agent will also wait for an activation message, which contains all the information for creating the appropriate probing process.

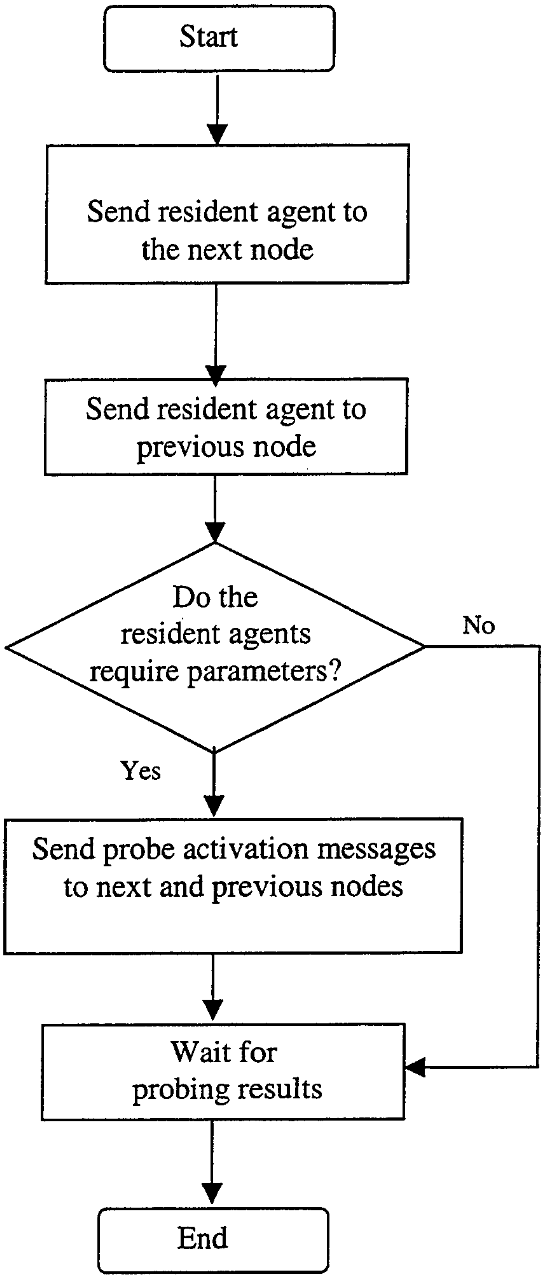

Start
Send resident agent to the next node
Send resident agent to previous node
Do the resident agents require parameters?
No
Yes
Send probe activation messages to next and previous nodes
Wait for probing results
End

Figure 20. Flow Chart of Probing Process on the Server Side

The *previous node* receives the resident agent, and waits for the *activation message*, that will initiate the probing process. The flow chart reflects the overall process on the *previous node* side.

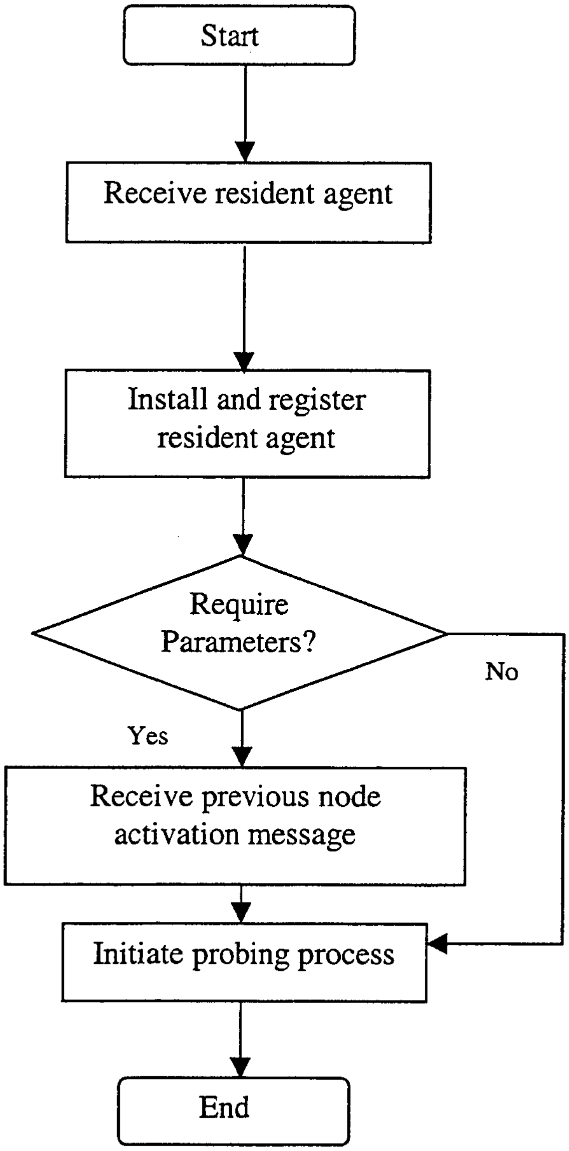


Figure 21. Flow Chart of Probing on the Previous Node Side

The *next node* receives the resident agent and waits for the activation message that will initiate the probe monitoring and computation process. The flow chart reflects the entire process on the *next node* side.

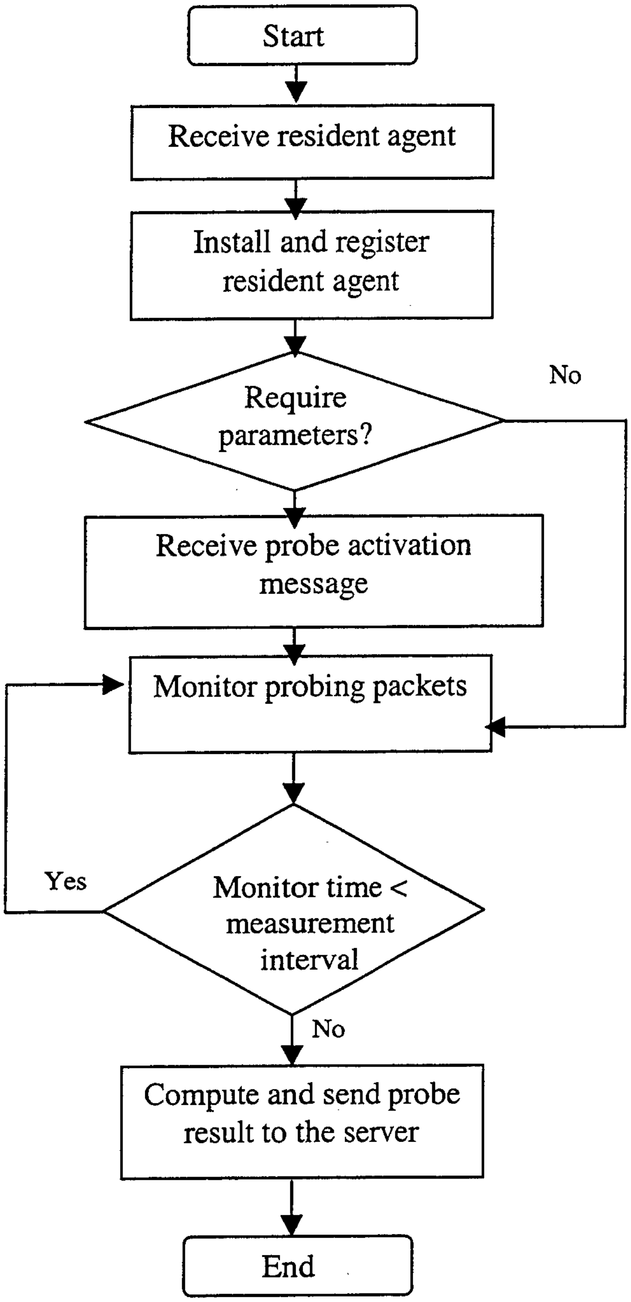

Start
Receive resident agent
Install and register resident agent
Require parameters?
No
Receive probe activation message
Monitor probing packets
Yes
Monitor time < measurement interval
No
Compute and send probe result to the server
End

Figure 22. Flow Chart of Probing on the Next Node Side

The timing diagram of the overall SAAM server probing is shown in Figure 23. The resident agent needs to be sent first to the *next node* and then to the *previous node* to make sure that the *next node resident agent* is ready to monitor probing packets. Then the activation messages can be sent respectively.

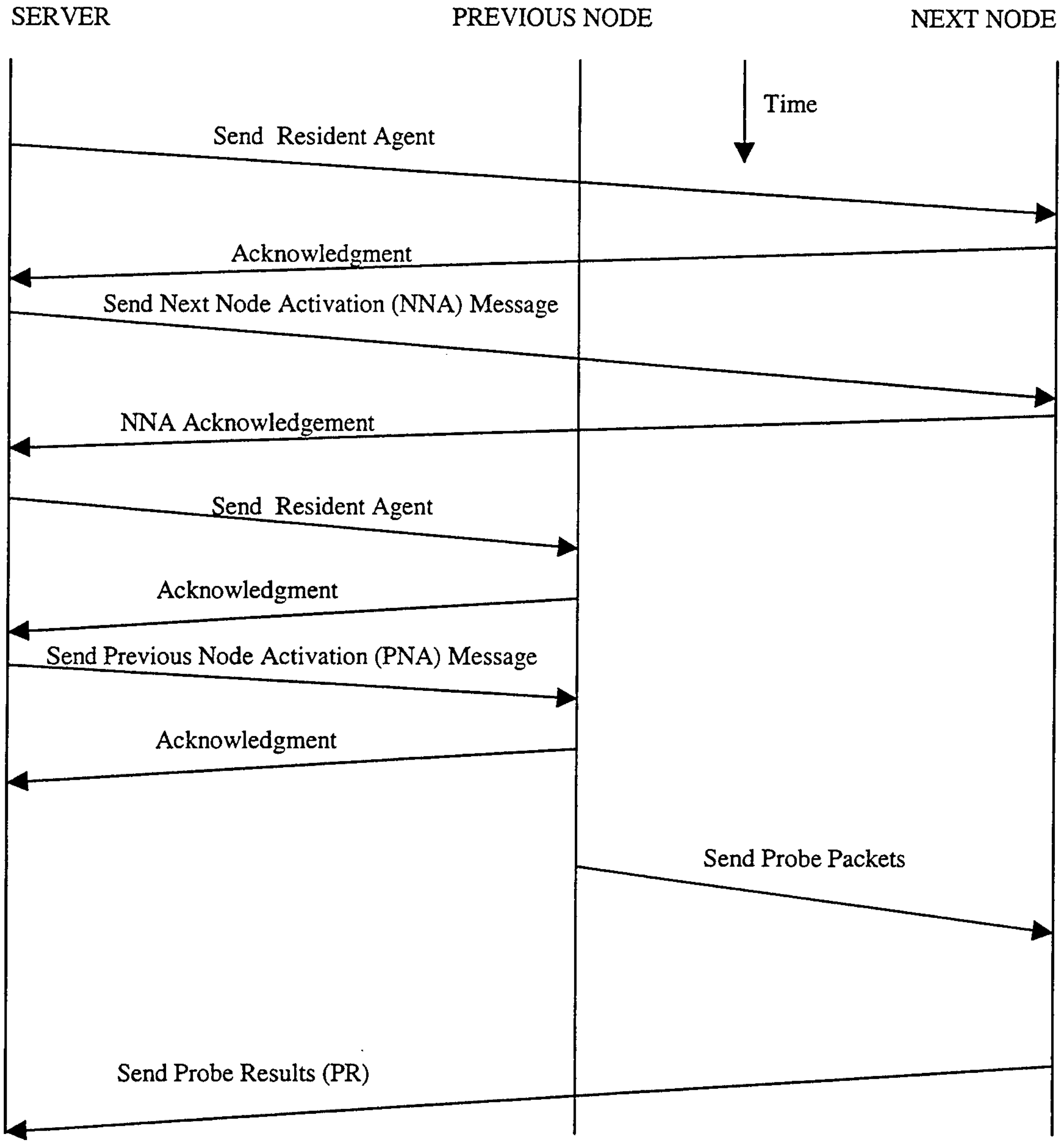


Figure 23. Timing Diagram of the Server Probing

THIS PAGE INTENTIONALLY LEFT BLANK

VI. IMPLEMENTATION OF SERVER PROBING

A working prototype of the proposed SAAM architecture was developed by two former Naval Postgraduate School students. The prototype emulates the routing of IPv6 packets within a SAAM network, and operates within the existing IPv4 network environment. Figure 24 illustrates the SAAM model with protocol layers.

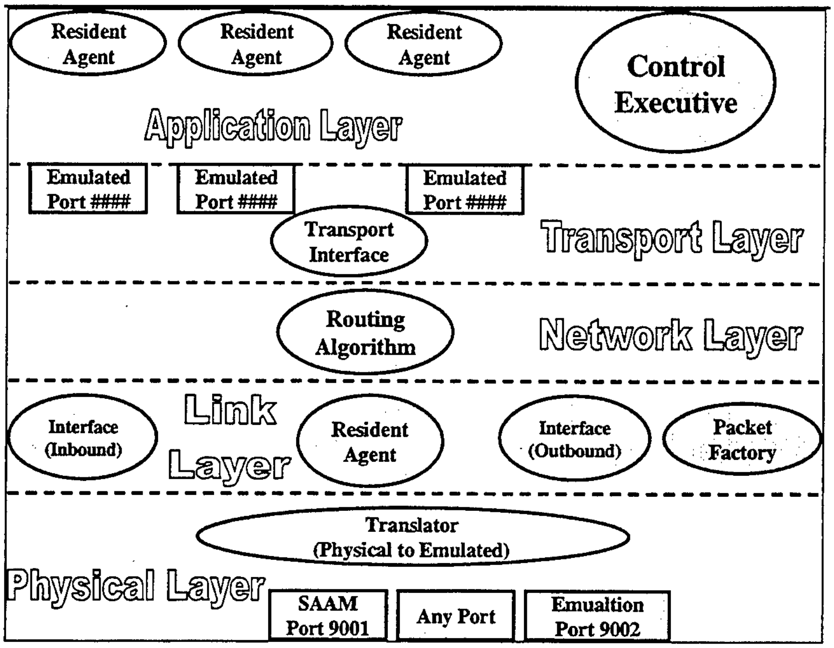


Figure 24. SAAM Prototype with Protocol Layers [From Ref.5]

In order to add server probing functionality to the SAAM prototype, some modifications to the existing code, including some additions to the class files, are proposed. These additions and modifications are based on the research findings of our studies. This chapter explains the added and modified class files, and the tests performed for this study.

A. GENERATION OF PROBE RELATED MESSAGES

1. PreviousNodeAct Class

The *PreviousNodeAct* class is an extension of the abstract *Message* class of the *saam.message* package. This class is used by the server to instantiate a *PreviousNodeAct* message. This message is sent to the *previous node* router to activate the probing agent installed on that router, and to provide the agent with a set of parameters specific to the current probe.

2. NextNodeAct Class

The *NextNodeAct* class is an extension of he abstract *Message* class of the *saam.message* package. The *saam.message* package is used by the server to instantiate a *NextNodeAct* message which is sent to the *next node* router to activate *the next node probing agent*. The *next node* router monitors the probing packets, calculates the probe results, and sends the results back to the server by instantiating a *ProbeResult* message

3. ProbeResult Class

The *ProbeResult* class is an extension of the abstract *Message* class of the *saam.message* package. This class is used by the *next node resident agent* to instantiate a probe result packet that will be sent to the server.

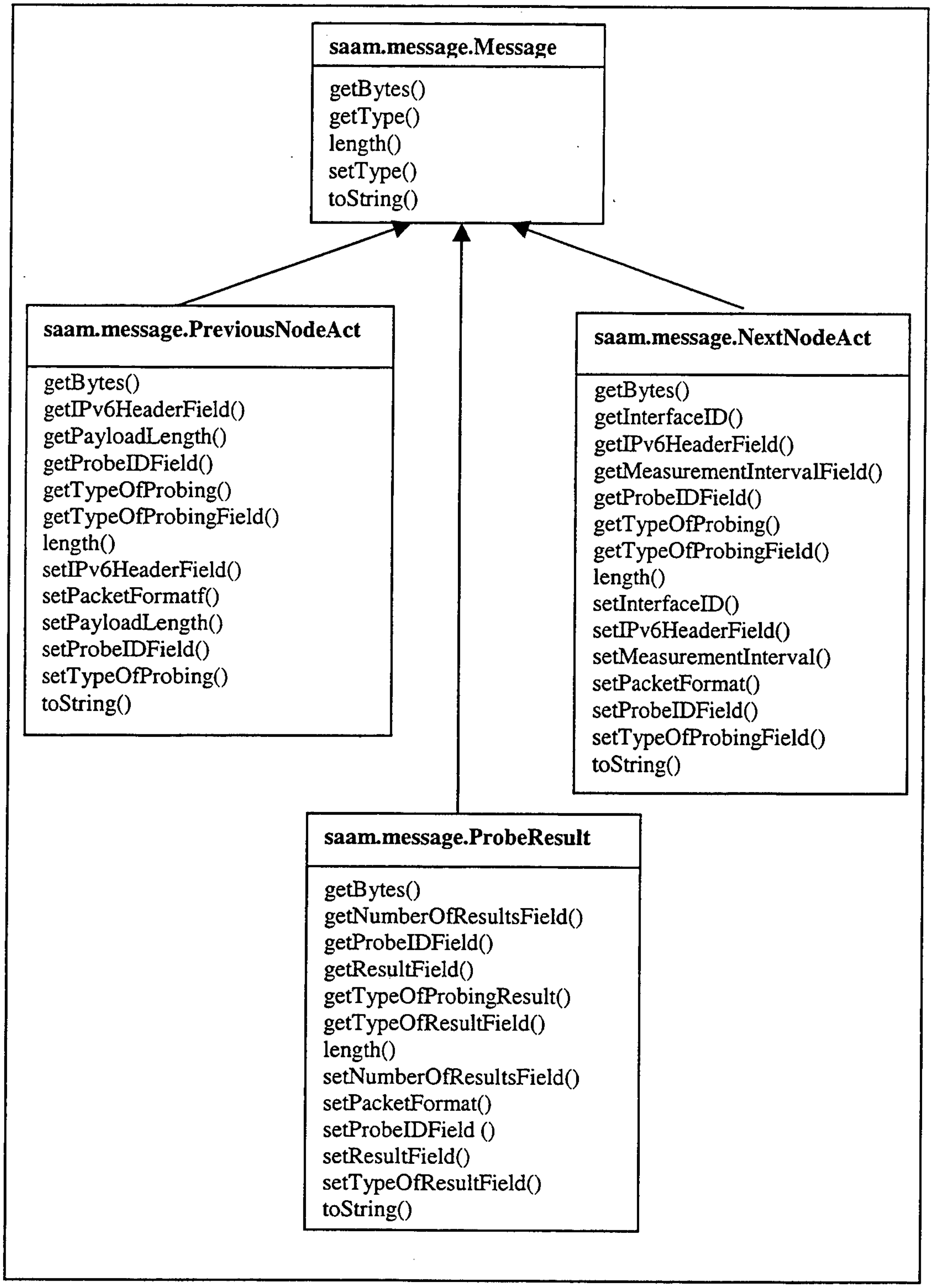


Figure 25. Message Class Structure

B. IMPLEMENTATION OF PROBING RESIDENT AGENTS

1. PreviousNodeProbe Resident Agent

A Java source file named "*PreviousNodeProbe.java*" is added to the *saam.residentagent.router* package, which includes all the resident agents that can be deployed to a router. The new file contains the code for the *PreviousNodeProbe class,* which is an extension of the Java *Thread* class. It implements the *ResidentAgentCustomer, ResidentAgent* and *MessageProcessor* classes of the *saam* package.

Implementing *ResidentAgentCustomer* allows the probing agent to register with the *ControlExecutive* as a dynamic router component. When a replacement of the probing agent arrives, the *ControlExecutive* calls the *replaceAgent()* method on all *customers* of the current *PreviousNodeProbe* agent. "

Implementing *ResidentAgent* provides the necessary calls for the *ControlExecutive (e.g., install, uninstall)*

Implementing *MessageProcessor* allows a probing resident agent to receive probe activation messages. (Recall that the server also sends a *PreviousNodeAct* message to the *previous node* to start the probing process.) Once a *PreviousNodeAct* message is received, the *ControlExecutive* will pass the message to the probing resident agent by calling the *processMessage()* method of that agent. The resident agent will parse the activation message and obtain the necessary parameters to start the probing process.

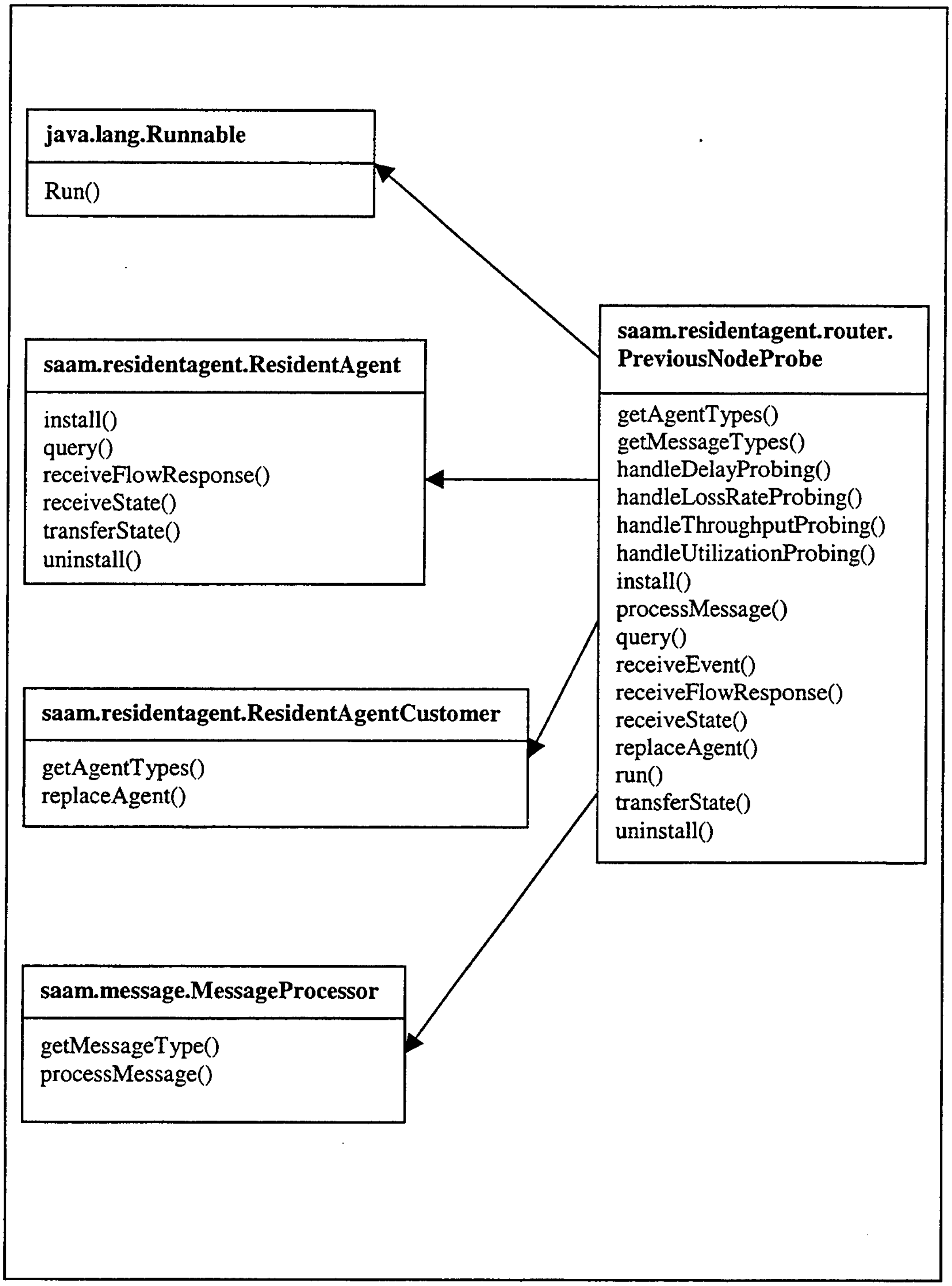


Figure26. PreviousNodeProbe Class Structure

2. NextNodeProbe Resident Agent

A Java source file named *"NextNodeProbe.java"* is added to the *saam.residentagent.router* package. The *NextNodeProbe* resident agent implements the *ResidentAgentCustomer*, *ResidentAgent*, *MessageProcessor*, and *SaamListener* class files.

One difference between the *NextNodeProbe* resident agent and the P*reviousNodeProbe* resident agent is the implementation of the *SaamListener* class. Components in the SAAM emulator can listen to particular *channels* to receive event notification by implementing the *SaamListener* class.

Channels are objects receiving events. Components registered to listen to a *channel* will receive an instance of each event sent to the *channel*. When a module wants to listen to a particular *channel*, the module has to register by using the *addListenerToChannel()* method in the *ControlExecutive* class.

Implementing *SaamListener* is required for the *NextNodeProbe* resident agent to monitor a particular network interface. The *interface ID* is specified in the *NextNodeAct* message, and the associated *channel ID* with the *interface ID* is provided by the *ControlExecutive*. The resident agent will receive a copy of every packet coming through this interface.

The implementation of the *ResidentAgentCustomer*, *ResidentAgent*, and *MessageProcessor* classes are the same as the *PreviousNodeProbe* resident agent. The *ResidentAgentCustomer* is required to receive agent updates. The *ResidentAgent* provides necessary calls for the *ControlExecutive* to perform install, uninstall, and state transfer of the resident agent. The implementation of the *MessageProcessor* is required for the resident agent to receive the *NextNodeAct* activation message from the *ControlExecutive*.

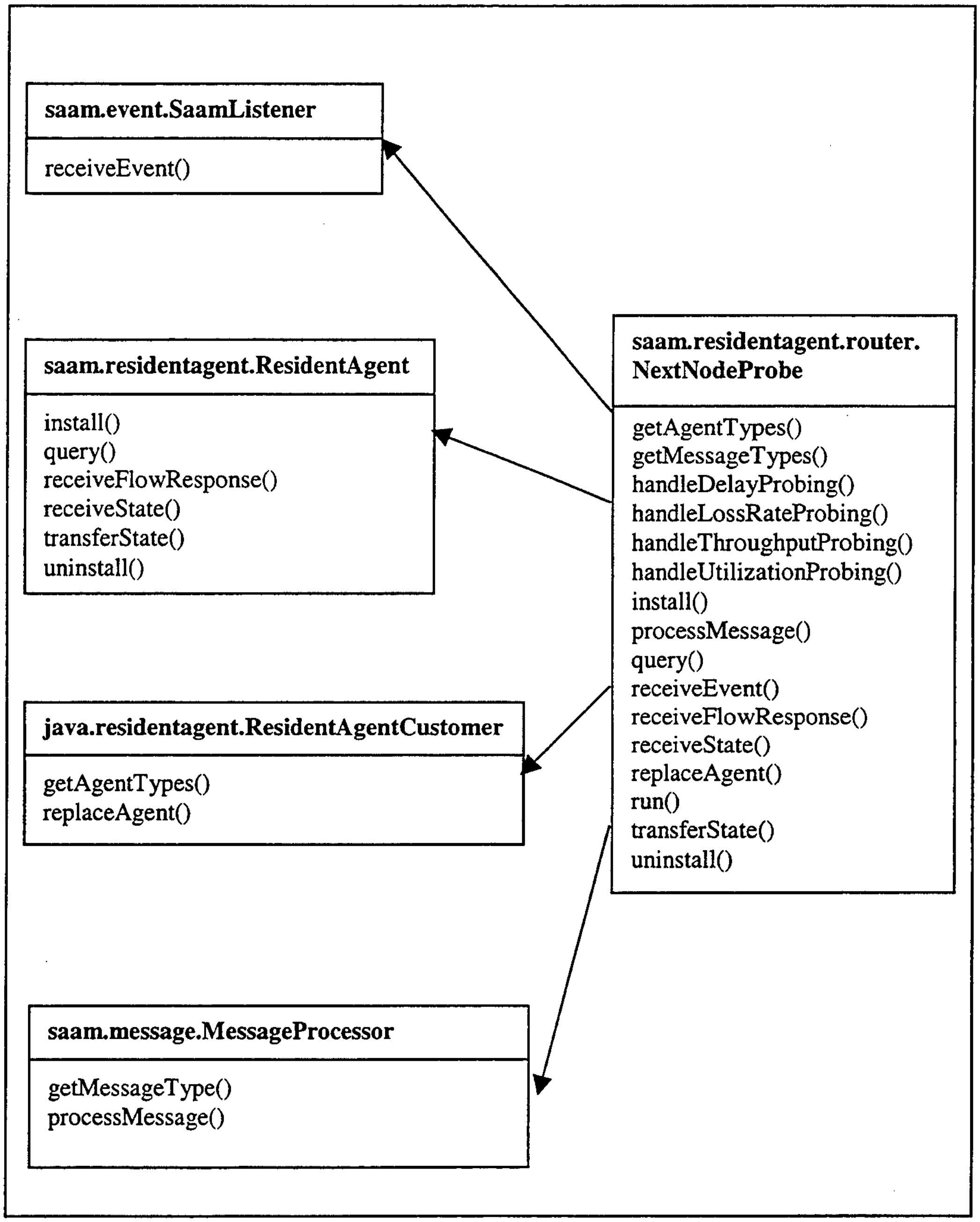


Figure27. NextNodeProbe Class Structure

C. MODIFICATIONS OF EXISTING CLASS FILES

1. PacketFactory Class File

The *PacketFactory* class is used by the *ControlExecutive* to send and receive SAAM control packets. The *PacketFactory* parses the content of each incoming packet to extract individual SAAM control messages. It then passes these messages to their corresponding message processors. For this task, several blocks of code have been added to the *processPacket()* method for handling of the probe activation and *ProbeResult* messages. The *append()* method has also been updated to include code for sending probe activation *ProbeResult* messages.

2. ServerAgent Class File

The *ServerAgent* class is a resident agent in the *saam.residentagent.server* package. A block of code has been added to its *processMessage()* method for handling of the *ProbeResult* messages. When *ServerAgent* receives a *ProbeResult* message, the *Server Agent* will pass the message to the *processProbeResult()* method of the *Server* class.

3. Server Class File

The *Server* class file is in the *saam.server* package. A method named *processProbeResult()* has been added to this class for processing of the *ProbeResult* messages. This method displays the probe result information sent by the *next node* resident agent.

In order to initiate the probing process from the server side, an *initiateProbing()* method is also added to the *Server* class file.

D. TEST RESULTS

The server probing code has been integrated into the SAAM prototype and tested with the network topology shown in Figure 26. In the test topology, two routers are set up as *previous node* and the *next node*. Although the test topology is simple, it provides a generic test topology. The described probing process is exactly the same for one or more routers between the *previous node* and the *next node*.

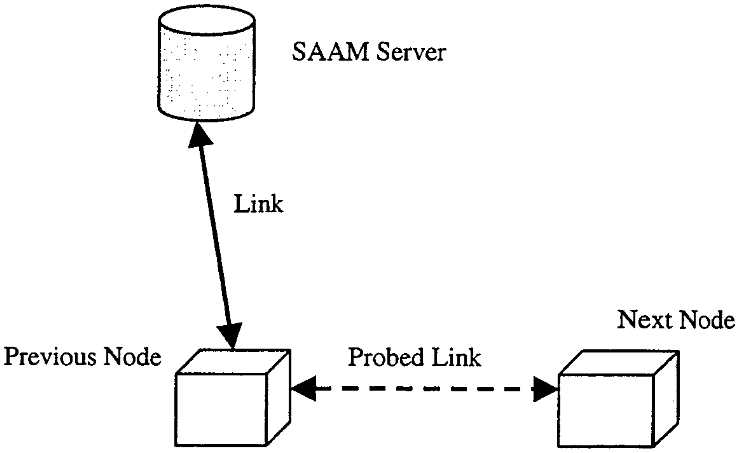
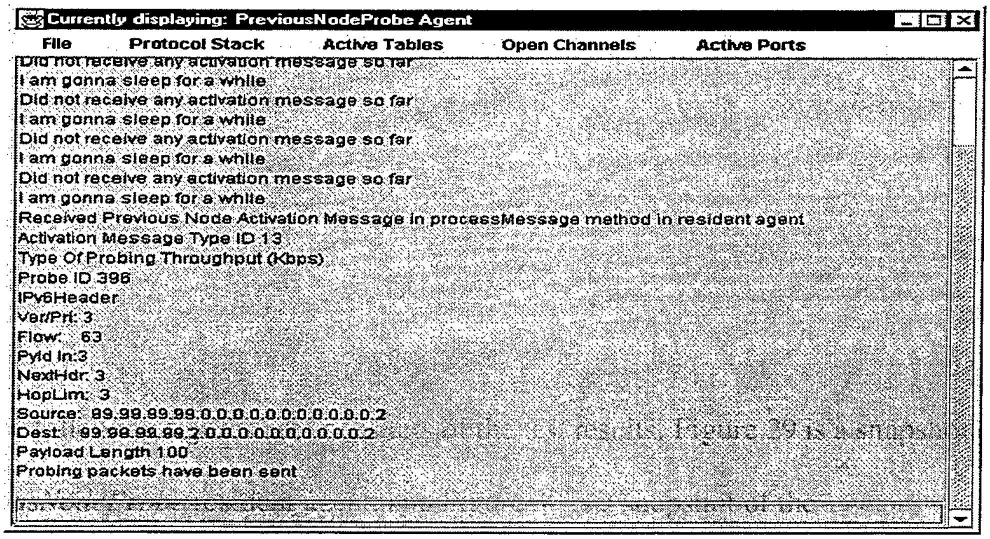


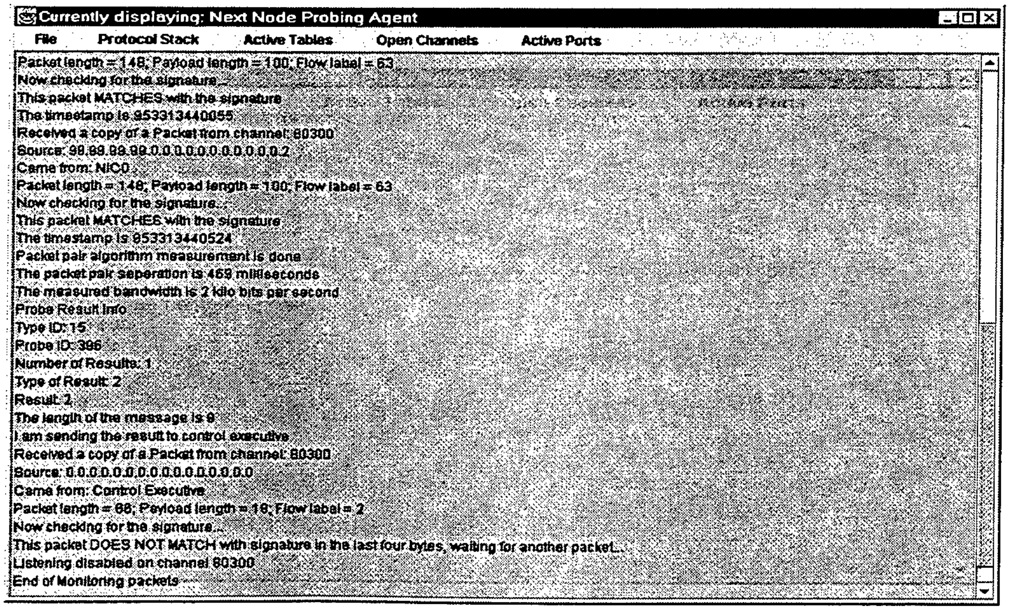
Figure 28. Server Probing Test Topology

The following figures are snapshots of the test results. Figure 29 is a snapshot of the *PreviousNodeProbe* resident agent, and Figure 30 is a snapshot of the *NextNodeProbe* resident agent. Finally, Figure 31 is a snapshot of the graphical user interface of the server displaying the *ProbeResult* message content.



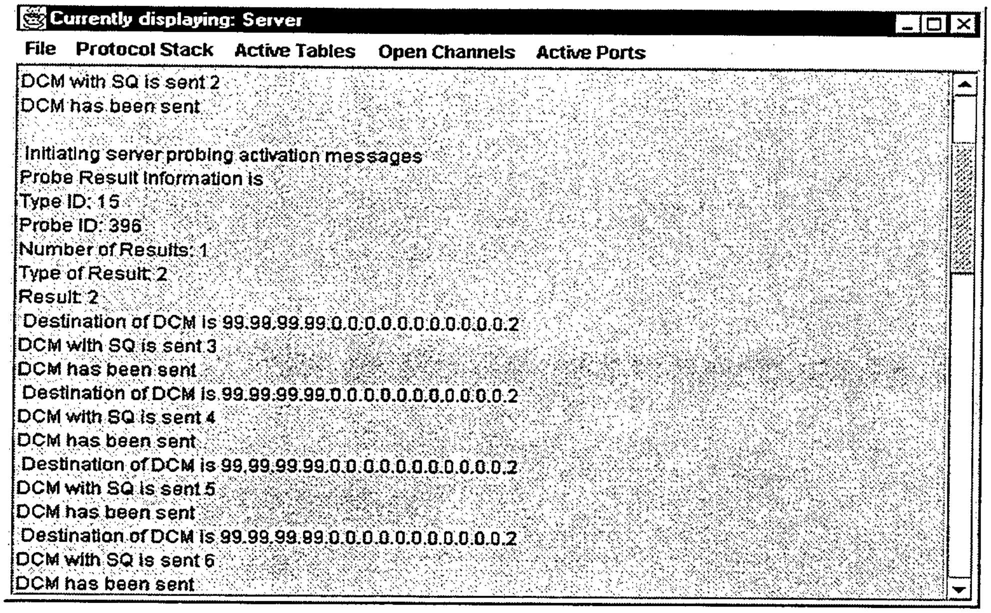
```
Did not receive any activation message so far
I am gonna sleep for a while
Did not receive any activation message so far
I am gonna sleep for a while
Did not receive any activation message so far
I am gonna sleep for a while
Did not receive any activation message so far
I am gonna sleep for a while
Received Previous Node Activation Message in processMessage method in resident agent
Activation Message Type ID 13
Type Of Probing Throughput (Kbps)
Probe ID 396
IPv6Header
Ver/Pri: 3
Flow:  63
Pyld ln:3
NextHdr: 3
HopLim: 3
Source: 99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.2
Dest: 99.99.99.99.2.0.0.0.0.0.0.0.0.0.0.2
Payload Length 100
Probing packets have been sent
```

Figure 29. Snapshot of PreviousNodeProbe Resident Agent



```
Packet length = 148; Payload length = 100; Flow label = 63
Now checking for the signature...
This packet MATCHES with the signature
The timestamp is 953313440055
Received a copy of a Packet from channel: 80300
Source: 99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.2
Came from: NIC0
Packet length = 148; Payload length = 100; Flow label = 63
Now checking for the signature...
This packet MATCHES with the signature
The timestamp is 953313440524
Packet pair algorithm measurement is done
The packet pair seperation is 469 milliseconds
The measured bandwidth is 2 kilo bits per second
Probe Result Info
Type ID: 15
Probe ID: 396
Number of Results: 1
Type of Result: 2
Result: 2
The length of the message is 9
I am sending the result to control executive
Received a copy of a Packet from channel: 80300
Source: 0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0
Came from: Control Executive
Packet length = 68; Payload length = 18; Flow label = 2
Now checking for the signature...
This packet DOES NOT MATCH with signature in the last four bytes, waiting for another packet...
Listening disabled on channel 80300
End of Monitoring packets
```

Figure 30. Snapshot of NextNodeProbe Resident Agent



```
DCM with SQ is sent 2
DCM has been sent

 Initiating server probing activation messages
Probe Result Information is
Type ID: 15
Probe ID: 396
Number of Results: 1
Type of Result: 2
Result: 2
 Destination of DCM is 99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.2
DCM with SQ is sent 3
DCM has been sent
 Destination of DCM is 99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.2
DCM with SQ is sent 4
DCM has been sent
 Destination of DCM is 99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.2
DCM with SQ is sent 5
DCM has been sent
 Destination of DCM is 99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.2
DCM with SQ is sent 6
DCM has been sent
```

Figure 31. Snapshot of the Server

A series of tests were performed to measure the throughput of the dotted link on Figure 26. Table 2 shows the measured time separations of the probing packets and the computed throughput of the dotted link on Figure 26.

Test Number	Time Separation Of The Two Probing Packets (ms)	Measured Throughput (Kbps)
1	16	74
2	485	2
3	63	18
4	15	78
5	437	2

Table 2. Test Results

The measured throughput values have a wide range, from 2 Kbps to 78 Kbps. The first reason for the unstable results is the operation of the SAAM prototype at the application layer. The prototype runs on a regular PC with many other processes running at the same time. The resource utilization of these processes fluctuates over time. The second reason is the lack of guaranteed CPU time for SAAM related Java threads.

VII. CONCLUSION

A. LESSONS LEARNED

A server probing model has been designed and developed for the SAAM architecture. The implementation has been tested after its integration into the current SAAM prototype. The developed server probing model provides a secure (in terms of not being detected by the *node being probed*) and practical measurement technique for the SAAM server.

1. Integration

The SAAM prototype implementation is getting bigger and bigger, and one of the most difficult part of this thesis was the integration process of the server probing code into the current SAAM prototype. Some unexpected code errors occurred, which were very difficult to debug without first acquiring a thorough knowledge of the existing system.

2. Coordination

Coordination of the SAAM developers is another important issue of this study. Since, every developer's work was inter-related with each other, this study required good coordination and frequent group meetings.

B. FUTURE WORK

1. Extending And Improving Server Probing

The current server probing implementation only allows the server to check the throughput metric of a particular service class for a particular link. It needs to be extended to include support for measurements of other metrics such as *packet loss rate* and *latency*. The framework for such extensions has been put in place and the probing resident agents are designed to be extensible. The most important part is to select an appropriate algorithm for each metric.

In the current implementation, the agents perform just one measurement. This needs to be improved so that the agents could perform a sequence of measurements and collect multiple samples for one probe. The average result could then be computed and sent to the server.

2. Adding Server Probing Intelligence to the SAAM Server

There are two important questions that need to be answered:

- In what circumstance will the SAAM server initiate the probing process?
- How will the server use the results?

The probed link is predetermined and is specified in the current server probing implementation. Instead of this predetermination, heuristics should be built into the server to determine when and which link to probe.

In the current implementation the server just displays the content of a received probe result message on the graphical user interface, without performing any comparison between the probe results and the LSA data from the same link. The server should perform the comparison and take appropriate measures when there is a significant discrepancy between the two data. What kind of measures to take is a future research topic.

3. Security and Policy Management

The Control Executive receives the probing resident agents. Before instantiating these agents, the Control Executive should perform policy and security check. A policy check would prevent agents from accessing the *Channel* class directly and circumventing the Channel registration process. A security check would provide an authentication of the resident agents.

4. Rerouting of The Flows Under Interface Failures

The SAAM server needs to reroute the flows that are affected when an interface fails. The server should also update the PIB by deactivating the paths that include a SLP of the failed interface.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. JDK INSTALLATION

JDK 1.1.*, is available from http://www.blackdown.org/java–linux /mirrors.html

For Java Development Kit installation, go to the URL above, and find the nearest mirror location. Find the package for i*86 platform, and download the file directly to the desired directory or move the file later to any other location. Once this is completed, follow the installation guideline provided for you. General guideline is provided in this section. Unpack the package using the Linux OS command:

```
$ tar zxvf filename.tr.gz
```

Point the PATH variable to the Java executables (e.g., java, javac, jar), which is under the */bin* directory. Here is an example of this process:

```
PATH=/your-directory/your-directory/jdk1.1.7/bin:$PATH
```

Also point your CLASSPATH variable to the file *classes.zip*, which is under the */lib* directory in the unpacked Java package. Don't forget to define the full path for the CLASSPATH environment, such as:

```
CLASSPATH.:/your-directory/your-directory/jdk1.1.7/lib/classes
```

Make sure, when typing *javac* or one of the Java executables in the command line, the system will recognize it. If not, reboot the system or re-do the instructions provided in the guideline. Another way to check if the system has set the CLASSPATH environment correctly, is by using the command:

```
printenv CLASSPATH
```

After running this command, one should be able to see the CLASSPATH variable pointing to the *classes.zip* file as an output in the command shell.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. PIZZA INSTALLATION

Pizza v. 0.39, is available at http://www.cis.unisa.edu.au/~pizza/Dist/

Since our test bed has Linux Operating System, we will pick the compressed Pizza distribution for the Unix platform. Download that version to the desired directory or move the compressed file (pizzadist.tar.gz) later on.

Once the downloaded file is unpacked, that file will create a pizza directory with two subdirectories, which are *classes* and *src.* The next step to follow is including the *pizza/classes* subdirectory in the CLASSPATH as shown below.

CLASSPATH.:/your-directory/your-directory/pizza/compiler/Main.class

If you have a CLASSPATH variable defined, then one should simply append the directory structure shown above:

CLASSPATH.:/your-directory-path/pizza/compiler/Main.class

We also need a script file *pc* including the following line.

java –ms8m pizza.compiler.Main

This file can be saved in any directory, where CLASSPATH variable points. Or one can add the full directory path onto the CLASSPATH variable. For the sake of clarity, simply pick the first choice.

If something does not work properly, most likely it might be a problem related to the CLASSPATH variables. Make sure they are set.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. JAVACC INSTALLATION

JavaCC 0.6.1, is available at:

http://www.cis.upenn.edu/~switchware/PLAN/software.html

It is a fairly simple installation. Begin downloading the class file from the URL given above, which is approximately 550 Kb. After this is accomplished, run the class file by simply typing:

java JavaCC0_6_1

in a command shell. This process will extract the files, and will create four sub-directories, which are:

- *bin*
- *doc*
- *examples*
- *src*

The *bin* directory includes the *javacc*, *jjdoc*, and the *jjtree* executable files, which are relevant to our PLAN package. The *doc* directory has the installation and information files. The *examples* directory has some sample codes related with different topics. The source code of the *JavaCC* can be found in the *src* directory.

As stated previously, we are primarily interested in the executables in the *bin* directory. Before installing the PLAN package, include the *bin* sub-directory in your PATH as follows:

PATH".:/your-directory/JavaCC/ bin"

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D. OCAML INSTALLATION

Ocaml distribution is available at http://paullac.inria.fr/caml/ocaml/distrib.html. Since we were using Linux OS as the test bed of our research, we downloaded the version for the Linux platform. This is a compressed file, which has to be decompressed with the following syntax.

- *tar zxvf <filename>*

After decompressing the file, it is ready for the installation process. The sequence of installation is as follows:

- *./configure*
- *make world*
- *make bootstrap*
- *make opt*
- *make ocamlc.opt*
- *make ocamlopt.opt*
- *umask 022*
- *make install*
- *make clean*

Ocamlp4 is decompressed the same way. The installation process of it as follows:

- *./configure*
- *make world*
- *make bootstrap*
- *make opt*

- *make install*

One must apply patches, which are available at the PLAN home page. The first patch is needed for proper compilation. The syntax of the patch is as follows:

- *Patch –p1 < ocaml-2.02-patch1.diffs*

There is also another patch for Ethernet card access, and the syntax is as follows:

- *Patch –p1 < ocaml-patch-2.0*

After all these packages are installed, it is time for the installation of the PLAN distribution. First *PLAN-ocaml-3.2-src.tar.z* must be downloaded from the PLAN home page (http://www.cis.upenn.edu/~switchware/PLAN/). This file needs to be decompressed with the following syntax.:

- *uncompress PLAN-ocaml-3.2-src.tar.z*

After the decompressing process, go under the plan directory and type the following syntax:

- *make all*

This process will complete the installation procedure of PLAN 3.2 distribution written in Ocaml programming language.

APPENDIX E.   PLAN SERVER PROBING SOURCE CODE

```
(*
This program will create two packets at the destination node. This implements the Packet
Pair algorithm. The basic Packet Pair algorithm relies on the fact if two packets are
queued next to each other at the bottleneck link, they will exit the link t seconds apart:
t= packet size of ack / bandwidth
*)
(* the definitions for the service calls *)

svc print : ’a -> unit
svc getRB : void -> int
svc defaultRoute : host -> host * dev
svc thisHost: void -> host
svc gettimeofday: void -> int*int

(* the acknowledment which is sent back to the    node with the time stamp of the node
that is  sent from *)
fun ack( where:host, time:int*int )=
        (
        print( "Host is : " ); print ( where ); print( " Time is ");print(time ); print( "\n" )
        )
(* this is the main program that will generate the packet pair acknowledgments to the
sender node *)
fun packetpair(senderNode:host, previousNode:host, nextNode:host)=
    (
        print("Current Resource Bound is ");print(getRB());print("\n");
        print( RIPGetRoutes( ) );
        if(thisHost( ) = previousNode) then
        OnRemote( |sendpacket|(senderNode), nextNode, 50, defaultRoute )
        else();
        if(thisHost( ) = previousNode) then
        OnRemote( |sendpacket|(senderNode); nextNode, 50, defaultRoute )
        else();
        if( thisHost( ) <> nextNode ) then
        OnRemote( |packetpair|(senderNode, previousNode, nextNode ), previousNode,
        50, defaultRoute )
        else( )
    )
fun sendpacket( senderNode:host) =
    (
        let val record_time1: int * int = gettimeofday( ) in
        OnRemote( |ack|( senderNode, record_time1 ), senderNode, 30, defaultRout )
        End
    )
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX F. PREVIOUS NODE ACTIVATION MESSAGE SOURCE CODE

```
/**
 * File: PreviousNodeAct.java
 * Author: Mustafa Altinkaya
 *          Naval Postgraduate School
 * Contact: altinkaya@yahoo.com
 * Date: March 2000
 * Compiler: JDK 1.2.1
 */

package saam.message;

import saam.util.*;
import saam.net.*;

import java.net.UnknownHostException;

/**
 * After the server sends the probe initiating resident agent to the Previous
 * Node, the resident agent needs to know certain parameters to start the
 * probing process. This Previous Node Activation Message contains the necessary
 * parameters to activate the probing process. The message will contain the following
 * fields as the payload.
 *
 * Type ID (byte): The type ID for Previous Node Activation Message is 13.
 * Type Of Probing (byte): There are four probing types, DELAY=0, LOSS_RATE=1,
 *                      THROUGHPUT=2, UTILIZATION=3
 * Probe ID (int): The server will keep track of the probing ID, so it can determine
 *              what link/node is associated with which result.
 * IPv6Header (40 bytes): Since the server wants to probe a particular flow, it simply
 *                      sends a copy of the IPv6Header for that particular flow,
 *                      so the Previous Node can generate packets with this header
 *                      information.
 * Payload Length (short): This will determine the payload length of the probing packets.
 *                      Sending this information will prevent of generating fixed sized
 *                      probing packets. The Node Under Probe will not be able to
 *                      differentiate probing packets from the ordinary data packets.
 */
public class PreviousNodeAct extends Message{

        private byte [] message;
        private byte type_of_probing;
        private int probe_ID;
        private IPv6Header myIPv6;
private short payload_length;
public static final byte DELAY = 0;
public static final byte LOSS_RATE = 1;
public static final byte THROUGHPUT = 2;
public static final byte UTILIZATION = 3;

/**
 * The constructor with no parameters. Once the Previous Node Probing
 * resident agent is installed it needs activation message, which also
 * includes the necessary parameters.
 */
public PreviousNodeAct( ){
        super( Message.PREVIOUS_NODE_ACT );
}//end of constructor


/**
 * The constructor with a byte array as parameter. This assumes the
 * byte array is formed correctly, so it can extract the field
 * information properly.
 * @param packet The byte array containing the field information
 */
public PreviousNodeAct( byte [] packet ){
 super( Message.PREVIOUS_NODE_ACT );
 IPv6Header ipv6Header=null;
 message = null;
 this.message = packet;
 setTypeOfProbingField( packet[1] );
 setProbeIDField(
        PrimitiveConversions.getInt( Array.getSubArray(packet, 2, 6)) );
        try{
            ipv6Header = new IPv6Header( Array.getSubArray(packet, 6, 46));
        }catch( UnknownHostException uhe ){
                System.out.println( uhe.toString() );
        }
 setIPv6HeaderField( ipv6Header );
 setPayloadLength(
        PrimitiveConversions.getShort( Array.getSubArray(packet, 46, 48 )) );
}//end of constructor


/**
 * The constructor with the field information as parameters.
 * @param probeType There are four probing types, DELAY=0,
 * LOSS_RATE=1, THROUGHPUT=2, UTILIZATION=3
 * @param probeID This is assigned by the server, so it can keep track of various
 * probing
 * @param ipvv6Header This parameter will be used to form the probing packets
 * @param payloadLength This will determine the payload length of the probing
 * packets, which will provide dynamic probing packet length
 */
public PreviousNodeAct(byte probeType, int probeID,
                        IPv6Header ipv6Header, short payloadLength ){
        super( Message.PREVIOUS_NODE_ACT );
        setPacketFormat( probeType, probeID, ipv6Header, payloadLength );
}//end of constructor


/**
 * This method provides to set the fields to the desired values.
 * @param probeType There are four probing types, DELAY=0,
 * LOSS_RATE=1,  THROUGHPUT=2, UTILIZATION=3
 * @param probeID This is assigned by the server, so it can keep track of probing
 * @param ipvv6Header This parameter will be used to form the probing packets
 * @param payloadLength This will determine the payload length of the probing
 * packets, which will provide dynamic probing packet length
 */
 public void setPacketFormat( byte probeType, int probeID, IPv6Header
        ipv6Header, short payloadLength ){
        setTypeOfProbingField( probeType );
        setProbeIDField( probeID );
        setIPv6HeaderField( ipv6Header );
        setPayloadLength( payloadLength );
        message = Array.concat( Message.PREVIOUS_NODE_ACT, message );
         message = Array.concat( message, type_of_probing );
        message = Array.concat( message,
                PrimitiveConversions.getBytes( probe_ID ) );
        message = Array.concat( message, myIPv6.getHeader( ) );
        message = Array.concat( message,
                 PrimitiveConversions.getBytes( payload_length ) );
}//end of setPacketFormat() method


/**
 * This method will return the string representation of the probing type
 * @param probeType There are four probing types, DELAY=0,
 * LOSS_RATE=1, THROUGHPUT=2, UTILIZATION=3
 * @return The string representation of the probing type
 */
public String getTypeOfProbing( byte probeType){
        switch(probeType){
```

```
        case DELAY:
            return "Delay (ms)";
        case LOSS_RATE:
            return "Loss Rate (0.01%)";
        case THROUGHPUT:
            return "Throughput (Kbps)";
        case UTILIZATION:
            return "Utilization (0.01%)";
        default:
            return "ERROR: No such probing type";
    }//end of switch-case
}//end of getTypeOfProbing() method


/**
 * This method will return the value of "type of probing" field
 * @return Value of "type of probing" field
 */
public byte getTypeOfProbingField(){
    return type_of_probing;
}//end of getTypeOfProbingField() method


/**
 * This method will return the value of "probe ID" field
 * @return Value of "probe ID" field
 */
public int getProbeIDField(){
    return probe_ID;
}//end of getProbeIDField() method


/**
 * This method will return the IPv6Header object
 */
public IPv6Header getIPv6HeaderField(){
    return myIPv6;
}//end of getIPv6HeaderField() method


/**
 * This method return the value of the payload length field
 * @return The value of "payload length" field
 */
public short getPayloadLengthField(){
    return payload_length;
```

```
}//end of getPayloadLengthField() method


/**
 * This method will return the message format in byte array
 * representation.
 * @return Message format in byte array representation
 */
public byte [] getBytes() {
        return message;
}//end of getBytes() method


/**
 * This method will return the length of the messge
 * @return The length of the message
 */
public short length(){
        return (short)message.length;
}//end of length() method



/**
 * This method will set the probe type field with the passed parameter
 * @param probeType There are four probing types, DELAY=0,
 * LOSS_RATE=1,  THROUGHPUT=2, UTILIZATION=3
 */
public void setTypeOfProbingField( byte probeType ){
        type_of_probing = probeType;
}//end of setTypeOfProbingField() method


/**
 * This method will set the probe ID field with the passed parameter
 * @param probeID This identification will be assigned by the server, so
 *              it can keep track of various probings
 */
public void setProbeIDField( int probeID ){
        probe_ID = probeID;
}//end of setProbeIDField() method


/**
 * This method will set the IPv6Header field
 * @param ipv6Header This parameter will be used to generate the probing
```

```
     *             packet IPv6Header information
     */
    public void setIPv6HeaderField( IPv6Header ipv6Header ){
        myIPv6 = ipv6Header;
    }//end of setIPv6HeaderField() method


    /**
     * This method will set the payload length, which will prevent from generation
     * fixed sized probing packets
     * @param payloadLength It will determine the length of the probing packets
     */
    public void setPayloadLength( short payloadLength ){
        payload_length = payloadLength;
    }//end of setPayloadLength()


    /**
     * Returns a String representation of this Message.
     * @param none
    * @return String the String representation of this Message
    */
    public String toString(){
        return
            "\nType ID: "+Message.PREVIOUS_NODE_ACT+
            "\nType of Probing: "+type_of_probing+
            "\nProbe ID: "+probe_ID+
            "\nPayload Length: "+payload_length;
    }//end of toString() method

}//end of class file
```

APPENDIX G. NEXT NODE ACTIVATION MESSAGE SOURCE CODE

```
/**
 * File: NextNodeAct.java
 * Author: Mustafa Altinkaya
 *              Naval Postgraduate School
 * Contact: altinkaya@yahoo.com
 * Date: March 2000
 * Compiler: JDK 1.2.1
 */

package saam.message;

import saam.util.*;
import saam.net.*;

import java.net.UnknownHostException;


/**
 * After the server sends the Next Node Probing Agent to the Next Node, this
 * message will serve as an activation message, which also contains the parameters
 * the agent needs.The message will contain the following information as the
 * payload.
 * Type ID (byte): The type ID for this message type is 14.
 * Type of Probing (byte): This will determine what kind of probing will be initiated
 *                      by the Previous Node, so the Next Node Probing Agent can
 *                      do the computation according this information
 * Probe ID (int): This information will serve to set the Probe ID field when constructing
 *                 the Probe Result Message. It will also serve as providing information
 *                 to intercept probing packets. Because, the signature at the tail of
 *                 the probing packets is the same with the probe ID.
 * IPv6 Header (40 bytes): The IPv6 header information of the probing packets
 * Measurement Interval (short): This will determine for how long the resident agent
 *                              should monitor for probing packets.
 * NIC to be Monitored (byte): This will determine what NIC to monitor.
 *
 */
public class NextNodeAct extends Message{

        private byte [] message;
        private byte type_of_probing;
        private int probe_ID;
        private IPv6Header myIPv6;
private short measurement_interval;
private byte interfaceID;

public static final byte DELAY = 0;
public static final byte LOSS_RATE = 1;
public static final byte THROUGHPUT = 2;
public static final byte UTILIZATION = 3;


/**
 * This is the constructor with no parameters
 */
public NextNodeAct( ){
        super( Message.NEXT_NODE_ACT );
}//end of constructor


/**
 * The constructor with a byte array as parameter. This assumes
 * the byte is array is set correctly having the field information.
 * @param packet The byte array containing the required field info
 */
public NextNodeAct( byte [] packet ){
 super( Message.NEXT_NODE_ACT );
 IPv6Header ipv6Header=null;
 message = null;
 this.message = packet;
 setTypeOfProbingField( packet[1] );
 setProbeIDField(
        PrimitiveConversions.getInt(Array.getSubArray(packet, 2, 6)) );
        try{
                ipv6Header = new
                        IPv6Header( Array.getSubArray(packet, 6, 46));
        }catch( UnknownHostException uhe ){
                System.out.println( uhe.toString() );
        }//end try-catch
        setIPv6HeaderField( ipv6Header );
        setMeasurementInterval(
            PrimitiveConversions.getShort( Array.getSubArray(packet, 46, 48 )) );
        setInterfaceID( packet[48] );
}//end of constructor

/**
 * The constructor with the field parameters.
 * @param probeType There are four probing types, DELAY=0,
 * LOSS_RATE=1,THROUGHPUT=2, UTILIZATION=3
 * @param probeID This informatio will also be used by the Next Node Probing
 * Agent to identify the signature. It will also be used to construct
 * the Probe Result Message format
 * @param ipv6Header The header information of the probing packets
 * @param timeInterval It will be used by the Next Node Probing Agent for how
 * long to monitor probing packets
 * @param nicID The NIC where the agent will monitor for probing packets
 */
public void NextNodeAct( byte probeType, int probeID,
                IPv6Header ipv6Header, short timeInterval, byte nicID ){
        setPacketFormat( probeType, probeID, ipv6Header, timeInterval, nicID );
}//end constructor


/**
 * This method will set the message fields with the passed parameters
 * @param probeType There are four probing types, DELAY=0,
 * LOSS_RATE=1, THROUGHPUT=2, UTILIZATION=3
 * @param probeID This informatio will also be used by the Next Node Probing
 * Agent  to identify the signature. It will also be used to construct
 *  the Probe Result Message format
 * @param ipv6Header The header information of the probing packets
 * @param timeInterval It will be used by the Next Node Probing Agent for how
 * long to monitor probing packets
 * @param nicID The NIC where the agent will monitor for probing packets
 */
public void setPacketFormat( byte probeType, int probeID,
                IPv6Header ipv6Header, short timeInterval, byte nicID ){
 setTypeOfProbingField( probeType );
 setProbeIDField( probeID );
 setIPv6HeaderField( ipv6Header );
 setMeasurementInterval( timeInterval );
 setInterfaceID( nicID );

 message = Array.concat( Message.NEXT_NODE_ACT, message );
 message = Array.concat( message, type_of_probing );
 message = Array.concat( message, PrimitiveConversions.getBytes( probe_ID ) );
 message = Array.concat( message, myIPv6.getHeader( ) );
 message = Array.concat( message,
        PrimitiveConversions.getBytes( measurement_interval ) );
 message = Array.concat( message, interfaceID );
}//end of setPacketFormat() method


/**
 * The string representatio of the probing type
 * @param probeType There are four probing types, DELAY=0,
 * LOSS_RATE=1,THROUGHPUT=2, UTILIZATION=3
 */
public String getTypeOfProbing( byte probeType){
        switch(probeType){
                case DELAY:
                        return "Delay (ms)";
                case LOSS_RATE:
                        return "Loss Rate (0.01%)";
                case THROUGHPUT:
                        return "Throughput (Kbps)";
                case UTILIZATION:
                        return "Utilization (0.01%)";
                default:
                        return "ERROR: No such probing type";
        }//end of switch-case
}//end of getTypeOfProbing() method


/**
 * This method will return the value of the type of probing field
 * @return Will return the byte value of the type of probing field
 */
public byte getTypeOfProbingField(){
        return type_of_probing;
}//end of getTypeOfProbingField() method


/**
 * This method will return the value of the probe ID field
 */
public int getProbeIDField(){
        return probe_ID;
}//end of getProbeIDField() method


/**
 * This method will return the IPv6Header informatin field
 */
public IPv6Header getIPv6HeaderField(){
        return myIPv6;
}//end of getIPv6HeaderField() method


/**
```

```
 * This method will return the measurement interval field
*/
public short getMeasurementIntervalField(){
        return measurement_interval;
}//end of getMeasurementIntervalField() method


/**
 * This method will return the interface ID field
 */
public byte getInterfaceID(){
        return interfaceID;
}//end of getInterfaceID() method


/**
 * This method will return the message format as a byte array
 * @return Byte array representation of the message
 */
public byte [] getBytes() {
        return message;
}//end of getBytes() method


/**
 * This method will return the length of the message
 */
public short length(){
        return (short)message.length;
}//end of length() method


/**
 * This method will set the type of probing field with the passed
 * parameter.
 * @param probeType There are four probing types, DELAY=0,
 * LOSS_RATE=1, THROUGHPUT=2, UTILIZATION=3
 */
public void setTypeOfProbingField( byte probeType ){
        type_of_probing = probeType;
}//end of setTypeOfProbingField() method

/**
 * This method will set the probe ID field. The probe ID value
 * will be determined by the server, so it can keep track of probing
 * results.
```

```
     */
    public void setProbeIDField( int probeID ){
            probe_ID = probeID;
    }//end of setProbeIDField() method


    /**
     * This method will set the IPv6Header information of the message
     */
    public void setIPv6HeaderField( IPv6Header ipv6Header ){
            myIPv6 = ipv6Header;
    }//end of setIPv6HeaderField() method


    /**
     * This method will set the measurement interval field
     * with the passed parameter.
     * @param timeInterval This value will determine for how the
     * Next Node Probing Agent will monitor for probing packets
     */
    public void setMeasurementInterval( short timeInterval ){
            measurement_interval = timeInterval;
    }//end of setMeasurementInterval() method

    /**
     * This method will set NIC to be monitored field
     */
    public void setInterfaceID( byte nicID ){
            interfaceID = nicID;
    }//end of setInterfaceID() method


    /**
     * Returns a String representation of this Message.
     * @param none
     * @return String the String representation of this Message
     */
    public String toString(){
            return
                    "\nType ID: " + Message.NEXT_NODE_ACT+
                    "\nType of Probing: " + type_of_probing+
                    "\nProbe ID: " + probe_ID+
                    "\nMeasurement Interval: " + measurement_interval;
    }//end of toString() method

}//end of class file
```

APPENDIX H. PROBE RESULT MESSAGE SOURCE CODE

```
/**
 * File: ProbeResult.java
 * Author: Mustafa Altinkaya
 *       Naval Postgraduate School
 * Contact: altinkaya@yahoo.com
 * Date: March 2000
 * Compiler: JDK 1.2.1
 */

package saam.message;

import saam.util.*;
import saam.net.*;

/**
 * Once the Next Node collects the probing results,it needs to wrap up the information
 * and send it back to the SAAM server, so the server can do comparison with the LSA
 * messages. This class identifies the fields of the message, and wraps the probing results
 * in a nice format. The message will contain the following fields as the payload.
 *
 * Type ID (byte): The type ID for this message is 15
 * Probe ID (int): The probe ID, wich will determine for what link/node the result belongs
 * Number Of Result (byte): Will determine how many probing results the message
 * contains
 * Type Of Result (byte): Will determine what type the probing result is.There are four
 *                            probing result types, DELAY=0, LOSS_RATE=1,
 *THROUGHPUT=2, UTILIZATION=3.
 * Result (short): This is the value of the measurement result.
 *
 * For a single probing result the message length will be 9.
 */
public class ProbeResult extends Message{


        private byte [] message;
        private int probe_ID;
        private byte number_of_result;
        private byte type_of_result;
        private short result;

        public static final byte DELAY = 0;
        public static final byte LOSS_RATE = 1;
public static final byte THROUGHPUT = 2;
public static final byte UTILIZATION = 3;

/**
 * The constructor of this class with no parameters. After the SAAM server
 * initiates the probing, and once the measurement is done by the Next Node
 * it will wrap the probing result information to a message format, having
 * the probeID, number of results, type of results, and probing results fields.
 */
public ProbeResult( ){
        // The static variable PROBE_RESULT has the byte value 15
        super( Message.PROBE_RESULT );
}//end of constructor


/**
 * The constructor of this class with byte array parameter. The byte array must
 * be in the required format with the appropriate fields set correctly.
 * @param packet This is the byte array consisting of the necessary fields.
 */
public ProbeResult( byte [] packet){
        // The static variable PROBE_RESULT has the byte value 15
        super( Message.PROBE_RESULT );
        message = null;
        this.message = packet;
        setProbeIDField(
                PrimitiveConversions.getInt(Array.getSubArray(packet, 1, 5)) );
        setNumberOfResultsField( packet[5] );
        setTypeOfResultField( packet[6] );
        setResultField(
         PrimitiveConversions.getShort( Array.getSubArray(packet, 7, 9 )) );
}//end of constructor

/**
 * The constructor of this class with the required parameters to set the fields
 * appropriately.
 * @param probeID This is the identification number sent by the server. When the
 * next node will the probing result back to the server, it will set this parameter
 * so the server can identify for wich link/node the probing result belongs.
 * @param numberOfResult This message format can contain more than one
 *              probing results. This field will determine how many results there
 *              are in this message format.
 * @param typeOfResult It will determine what kind of probing result this
 * message contains. There are four probing result types, DELAY=0, * *
 *LOSS_RATE=1, THROUGHPUT=2, UTILIZATION=3.
 * @param result The probing result for a particular probe identification
 */
public ProbeResult( int probeID,
        byte numberOfResult, byte typeOfResult, short result ){
        setPacketFormat( probeID, numberOfResult, typeOfResult, result );
}


/**
 * This method will set the fields of the ProbeResult object
 * @param probeID This is the identification number sent by the server.
 * When the next node will the probing result back to the server, it will set this
 * parameter so the server can identify for which the probing result belongs.
 * @param numberOfResult This message format can contain more than one
 * probing results. This field will determine how many results there are in this
 * message format.
 * @param typeOfResult It will determine what kind of probing result this
 * message contains.
 * @param result The probing result for a particular probe identification
 */
public void setPacketFormat( int probeID,
                byte numberOfResult, byte typeOfResult, short result ){
        setProbeIDField( probeID );
        setNumberOfResultsField( numberOfResult );
        setTypeOfResultField( typeOfResult );
        setResultField( result );

        message = Array.concat( Message.PROBE_RESULT, message );
        message = Array.concat( message,
                PrimitiveConversions.getBytes( probe_ID ) );
        message = Array.concat( message, number_of_result );
        message = Array.concat( message, type_of_result );
        message = Array.concat( message,
                PrimitiveConversions.getBytes( result ) );
}//end of setPacketFormat() method


/**
 * This method will return the type of probing in a string format
 * @param probeType The probe type in byte format.
 * @return Will return the probe type in string format.
 */
public String getTypeOfResult( byte resultType){
        switch(resultType){
                case DELAY:
                        return "Delay (ms)";
                case LOSS_RATE:
```

```
            return "Loss Rate (0.01%)";
        case THROUGHPUT:
            return "Throughput (Kbps)";
        case UTILIZATION:
            return "Utilization (0.01%)";
        default:
            return "ERROR: No such probing type";
    }//end of switch-case
}//end of getTypeOfProbing() method


/**
 * This method will return the int value of the probe identification field.
 * @return The int value of probe identification,which is four bytes long
 */
public int getProbeIDField(){
    return probe_ID;
}//end of getProbeIDField() method


/**
 * This method will return the byte value of the number of results field.
 * @return The byte value of number of results field, which is 1 byte long
 */
public byte getNumberOfResultsField(){
    return number_of_result;
}//end of getNumberOfResultsField() method


/**
 * This method will return the byte value of the type of result field.
 * @return The byte value of the type of result field, which is 1 byte long.
 */
public byte getTypeOfResultField(){
    return type_of_result;
}//end of getTypeOfResultField() method


/**
 * This method will return the short value of the result field.
 * @return The short value of the result field which is 2 bytes long.
 */
public short getResultField( ){
    return result;
}//end of getResultField() method
```

```
/**
 * This method will return the Probe Result message format as a byte array.
 * @return The message format as a byte array.
 */
public byte [] getBytes() {
        return message;
}//end of getBytes() method


/**
 * This method will return the length of the message format payload,which is 9 for
 * one probing result. The length will depend on the number of probing results
 * the message contains.
 * @return The length of the Probing Result message
 */
public short length(){
        return (short)message.length;
}//end of length() method


/**
 * This method will set the probe identification field with the passed parameter.
 * @param probeID The probe identifiction value.
 */
public void setProbeIDField( int probeID ){
        probe_ID = probeID;
}//end of setProbeIDField() method


/**
 * This method will set the number of results field with the passed parameter.
 * @param numberOfResult Will identify the number of results
 */
public void setNumberOfResultsField( byte numberOfResult ){
        number_of_result = numberOfResult;
}//end of setNumberOfResultsField() method


/**
 * This method will set the type of result field with the passed parameter.
 * @pram typeOfResult DELAY=0, LOSS_RATE=1,
 *              THROUGHPUT=2, UTILIZATION=3.
 */
public void setTypeOfResultField( byte typeOfResult ){
        type_of_result = typeOfResult;
```

```
    }//end of setTypeOfResultField() method


    /**
     * This method will set the probing result field with the passed parameter.
     * @param result The result value.
     */
    public void setResultField( short result ){
            this.result = result;
    }//end of setResultField() method


    /**
     * Returns a String representation of this Message.
     * @param none
     * @return String the String representation of this Message
     */
    public String toString(){
            return
                    "\nType ID: " + Message.PROBE_RESULT+
                    "\nProbe ID: " + probe_ID+
                    "\nNumber of Results: " + number_of_result+
                    "\nType of Result: " + type_of_result+
                    "\nResult: " + result;
    }//end of toString() method

}//end of class file
```

APPENDIX I. PREVIOUS NODE PROBING RESIDENT AGENT SOURCE CODE

```
/**
 * File: PreviousNodeProbe.java
 * Author: Mustafa Altinkaya
 *          Naval Postgraduate School
 * Contact: altinkaya@yahoo.com
 * Date: March 2000
 * Compiler: JDK 1.2.1
 */

package saam.residentagent.router;

import saam.residentagent.*;
import saam.message.*;
import saam.control.*;
import saam.util.*;
import saam.event.*;
import saam.net.*;
import saam.router.*;

/**
 * A ResidentAgent is an Object that can be delivered accross a SAAM
 * network to a router, perform some type of processing or monitoring
 * on that router, and then be replaced, forward its state to another
 * router, or uninstall itself. In this case the Previous Node Probing
 * Resident Agent will initiate the probing process. Since the Activation
 * Message will be sent later by the server, this resident agent is the
 * message processor for the PreviuosNodeAct type message. Once the it
 * resceives that message it can start the probing process. This resident
 * agent is capable of probing for DELAY, LOSS RATE, THROUGHPUT and
 * UTILIZATION.Packet pair algorithm is used for THROUGHPUT probing. The other
 * probings are left for future work, but the groundwork has been provided.
 */

public class PreviousNodeProbe extends Thread implements
        ResidentAgentCustomer,ResidentAgent,MessageProcessor {

        private ControlExecutive controlExec;
        private SAAMRouterGui gui;
        private String [] messageTypes = { "saam.message.PreviousNodeAct"};
        private static final String[] agentTypes =
                {"saam.residentagent.router.PreviousNodeProbe"};
         private byte [] v6payloadtemp;
 private byte [] v6payload;
private byte [] dummyUDParray;
private UDPHeader udp;
private IPv6Packet v6packet;
private boolean received=false;
private PreviousNodeAct pna=null;


/**
* Within this method, an agent provides the necessary calls to the
* ControlExecutive that performs all necessary registration.
* @param controlExec The ControlExecutive on the router this agent
*      is being installed on.
*/
public void install( ControlExecutive controlExec ){

        gui = new SAAMRouterGui( "PreviousNodeProbe Agent" );
        this.controlExec = controlExec;
        // This resident agent is the message processor for PreviuousNodeAct type
        // activation messages.
        controlExec.registerMessageProcessor(this);
        controlExec.registerCustomer(this);
        gui.sendText("Installation of Previous
                        Node Probing Agent is completed");
        start();
}



/**
 * This method is called right after the start() method, and runs as a
 * seperate thread. This has been used for SAAM configuration settlement
 * purposes.
 */
public void run(){
        //if still the activatio message has not been received, go into another
        // sleep cycle.
        while(!received){
                try{
                        gui.sendText("I am gonna sleep for a while ");
                        this.sleep(10000);
                }catch( InterruptedException ie){
                        gui.sendText(ie.toString() );
                }//try-catch
        // if the activation message has been received, and constructed succesfully
        // switch to the appropriate method, depending
```

```
    // on the type of probing field.
    if(received){
        switch( pna.getTypeOfProbingField() ){
        // This will initiate DELAY probing
            case PreviousNodeAct.DELAY :
                handleDelayProbing();
            break;
        // This will initiate LOSS RATE probing
            case PreviousNodeAct.LOSS_RATE:
                handleLossRateProbing();
            break;
        // This will initiate THROUGHPUT probing
            case PreviousNodeAct.THROUGHPUT:
                handleThroughputProbing( pna.getProbeIDField(),
                pna.getIPv6HeaderField(),
                (int)pna.getPayloadLengthField());
            break;
        // This will initiate UTILIZATION probing
            case PreviousNodeAct.UTILIZATION:
                handleUtilizationProbing();
            break;
            default:
            gui.sendText("ERROR: Unrecognized Probing Type");
        }//switch-case
    }else{
    gui.sendText("Did not receive any activation message so far " );
    }// if-else
    }//end while
}//end of run() method

/**
 * The resident agent is also registered as a message processor for
 * PreviousNodeAct type messages. Once the router receives this message type,
 * the control executive will demultiplex the message to the registered customer.
 * @param message The subclass of saam.message.Message to be processed.
 */
public void processMessage( Message message ){
    try{
        if( message instanceof PreviousNodeAct){
            pna = ( PreviousNodeAct ) message;
            v6payloadtemp = new
                byte[ (int)pna.getPayloadLengthField() ];
            gui.sendText("Received Previous Node
                Activation Message in processMessage method in
                resident agent ");
            gui.sendText("Activation Message Type ID "
                        + Message.PREVIOUS_NODE_ACT );
                gui.sendText("Type Of Probing "+
                pna.getTypeOfProbing( pna.getTypeOfProbingField()) );
                gui.sendText("Probe ID " + pna.getProbeIDField() );
                gui.sendText("IPv6Header " +
                        pna.getIPv6HeaderField().toString() );
                gui.sendText("Payload Length " +
                        pna.getPayloadLengthField() );
                received = true;
            }//end if
        }catch( Exception e){
            message = null;
        }//try-catch
}//end of processMessage() method


/**
 * This part of the probing is left as future work. Any algorithm regarding
 * for DELAY parameter probing can be implemented in this method, as it
 * is done for the throughput probing using modified packet pair algorithm.
 */
public void handleDelayProbing(){
        //future work
}


/**
 * This part of the probing is left as future work. Any algorithm regarding
 * for LOSS RATE parameter probing can be implemented in this method, as it
 * is done for the throughput probing using modified packet pair algorithm.
 */
public void handleLossRateProbing(){
        //future work
}


/**
* The modified packet pair algorthm is implemented in this method. To be able
* to identify probing packets from regular data packets the payload contains a
* signature which is 4 bytes long at the end of the packet. The signature is the
* same with the probe ID. Since the Next Node Probing Agent will also receive
* the probe ID,it will know what the signature is, and it will do the comparison
* with the probe ID.This method will generate to packets according to the passed
* parameters, and it will be handed over to the control executive to be processed
* and send out to its destination address.
* @param probeIdentification This will be used as the signature at the tail of
*            the probing packets
* @param v6Header The IPv6Header information to construct the probing
* packets. This will also determine for what flow the probing will be done.
* @param length This will determine the length of the probing packets. Sending
* the size of the probing packets as a parameter will prevent of sending
*            fixed sized probing packets.
*/
public void handleThroughputProbing(int probeIdentification,
                IPv6Header v6header, int length){
        // construct the payload of the probing packet, 4 bytes less than the
        // passed legth, so the signature can be placed at the tail of the payload
        v6payload = new byte [length - 4];
        // now append the signature to the tail, so the total length of the packet
        // will be the size of the passed length
        v6payload = Array.concat( v6payload,
                PrimitiveConversions.getBytes(probeIdentification));
        dummyUDParray = new byte[8];
        udp = new UDPHeader( dummyUDParray );
        v6packet = new IPv6Packet( v6header, udp, v6payload );
        try{
                // the for loop will send 2 probing packets
                for(int i=1;i<=2;i++){
                        //now hand the packet to the control executive, so it can
                        //take care of the rest of the sending process
                        controlExec.send( this, v6packet );
                }// end for loop
        }catch(FlowException fe){
                gui.sendText("Flow Exception occured ");
        } //try-catch
                gui.sendText("Probing packets have been sent");
        }//end of handleThroughputProbing() method


/**
* This part of the probing is left as future work. Any algorithm regarding
* for UTILIZATION parameter probing can be implemented in this method, as it
* is done for the throughput probing using modified packet pair algorithm.
*/
public void handleUtilizationProbing(){
        //future work
}//end of handleUtilizationProbing() method



/**
* This method will simply return the agent types in string format.
```

```
*/
public String[] getAgentTypes(){
        return agentTypes;
}//getAgentTypes()


/**
* When the ControlExecutive receives an agent that is to replace
* an existing agent, the ControlExecutive calls the replaceAgent
* method on each of the ResidentAgentCustomers of that ResidentAgent.
* @param replacement The new ResidentAgent.
*/
public synchronized void replaceAgent( ResidentAgent agent){
}//replaceAgent()


/**
* When an agent is being replaced, the ControlExecutive will remove
* the old agent from all channels it is allowed to talk on and from
* all channels it is monitoring.  The uninstall method is called by
* the ControlExecutive upon replacement in order to allow the old
* agent a chance to perform any other cleanup that might be necessary,
* such as disposing of a user interface, for instance.
*/
public void uninstall(){}


/**
* When an agent is about to be replaced, the ControlExecutive calls
* the transferState method, passing the old agent a copy of the new
* agent.  The old agent should then call the receiveState method on
* the new agent, and pass to the new agent any messages that should
* be passed.
* @param replacement The agent that is replacing the old agent.
*/
public void transferState( ResidentAgent replacement ){}


/**
* This method is called one or more times by an agent that is about
* to be replaced by this agent.  The purpose of this method is to
* enable a state transfer from the old agent to the new agent.
* @param message The message to be passed from the old agent to the
*       new agent.
*/
public void receiveState(Message message){}
```

```
    /**
    * When a ResidentAgent requests a flow by calling the requestFlow method
    * of the ControlExecutive, the agent expects to be assigned a flow and
    * to be notified when that flow is assigned.  This method provides a
    * mechanism for notifying the ResidentAgent that a FlowResponse has
    * arrived.
    */
    public void receiveFlowResponse(FlowResponse flowResponse){ }

    /**
    * Some resident agents are accessed by Objects on the router.  This
    * method provides the means for communication between an Object on
    * the router and this ResidentAgent.
    * @param message The Message the Object sends to this ResidentAgent.
    * @return The Message this ResidentAgent sends back to the Object
    *         performing the query.
    */
    public Message query( Message message ){ return message; }


    /**
    * Returns an array that contains the class names of the resident
    * agents that this Object desires to be a ResidentAgentCustomer of.
    * @return An array that contains the class names of the resident
    * agents that this Object desires to be a ResidentAgentCustomer of.
    */
    public String[] getMessageTypes(){ return messageTypes;}


    /**
    * This method is called by the Channel to pass event notifications
    * to the Objects.
    */
    public void receiveEvent(SaamEvent se){ }
}//end of class file
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX J. NEXT NODE PROBING RESIDENT AGENT SOURCE CODE

```
/**
 * File: NextNodeProbe.java
 * Author: Mustafa Altinkaya
 *           Naval Postgraduate School
 * Contact: altinkaya@yahoo.com
 * Date: March 2000
 * Compiler: JDK 1.2.1
 */
package saam.residentagent.router;

import saam.residentagent.*;
import saam.message.*;
import saam.control.*;
import saam.util.*;
import saam.event.*;
import saam.net.*;

import java.net.UnknownHostException;


/**
 * A ResidentAgent is an Object that can be delivered across a SAAM
 * network to a router, perform some type of processing or monitoring
 * on that router, and then be replaced, forward its state to another
 * router, or uninstall itself. In this case the NextNodeProbe resident
 * agent is in charge of monitoring the probing packets and compute the
 * probing parameters. There are four probing parameters, this resident
 * agent can compute the statistics of it. These are DELAY, LOSS RATE,
 * THROUGHPUT and UTILIZATION. The resident agent will wait for the
 * activation message which will include the parameters, type of probing
 * probe ID, IPv6Header information, measurement interval, and the NIC
 * identification to be monitored for probing packets. It will monitor
 * the NIC, looking for the packets that match the signature which is
 * at the tail of the payload. For throughput measurement it will
 * intercept the two probing packets recording the time of arrival.
 * The throughput will be computed according the packet pair algorithm
 * with the following formula
 *
 *    throughput = (packet length)/(time separation of the two packets)
 */
```

```
public class NextNodeProbe implements ResidentAgentCustomer, SaamListener,
        ResidentAgent,MessageProcessor {

    private ControlExecutive controlExec;
    private SAAMRouterGui gui;
    private String [] messageTypes = { "saam.message.NextNodeAct"};
    private static final String[] agentTypes =
        {"saam.residentagent.router.NextNodeProbe"};
    private int channel_ID;
    private int signature;
    private NextNodeAct nna=null;
    private ProbeResult pr=null;
    private long [] timestamps;
    private int pointer = 0;
    private int bandwidth;
    private boolean received=false;


    /**
    * Within this method, an agent provides the necessary calls to the
    * ControlExecutive that performs all necessary registration.
    * @param controlExec The ControlExecutive on the router this agent
    *       is being installed on.
    */
    public void install( ControlExecutive controlExec ){
        gui = new SAAMRouterGui( "Next Node Probing Agent " );
        this.controlExec = controlExec;
        controlExec.registerMessageProcessor(this);
        controlExec.registerCustomer(this);
        gui.sendText("Installation of Next Node Probing Agent is completed ");
        timestamps = new long [2];
        pr = new ProbeResult();
    }


    /**
     * The resident agent is also registered as a message processor for NextNodeAct
     * type messages. Once the router receives this message type, the control
     * executive will demultiplex the message to this registered customer.
     * @param message The subclass of saam.message.Message to be processed.
     */
    public void processMessage( Message message ){
        try{
            // check if the message type is a Next Node Activatin one
            if( message instanceof NextNodeAct){
                nna = ( NextNodeAct ) message;
                gui.sendText("Received Next Node Activation Message ");
                gui.sendText("Next Node Activation Message Type ID is "
                        + Message.NEXT_NODE_ACT );
                gui.sendText("Type Of Probing is " +
                        nna.getTypeOfProbing(
                        nna.getTypeOfProbingField()) );
                gui.sendText("Probe ID is " + nna.getProbeIDField() );
                gui.sendText("IPv6Header information is " +
                        nna.getIPv6HeaderField().toString() );
                gui.sendText("Measurement Interval is " +
                        nna.getMeasurementIntervalField() );
                gui.sendText("Interface Identification is " +
                        nna.getInterfaceID());
        switch( nna.getTypeOfProbingField() ){
        //is the probing process for DELAY parameter
                case NextNodeAct.DELAY :
                        handleDelayProbing();
                break;
                //is the probing process for LOSS RATE parameter
                case NextNodeAct.LOSS_RATE:
                        handleLossRateProbing();
                break;
                //is the probing process for DELAY parameter
                case NextNodeAct.THROUGHPUT:
                        handleThroughputProbing( nna.getProbeIDField(),
                        nna.getIPv6HeaderField(),
                        nna.getMeasurementIntervalField(),
                        nna.getInterfaceID());
                break;
                //is the probing process for DELAY parameter
                case NextNodeAct.UTILIZATION:
                        handleUtilizationProbing();
                break;
                default:
                        gui.sendText("ERROR:
                                Unrecognized Probing Type");
        }//end switch-case
        }//end if
    }catch( Exception e){
        message = null;
    }//end try-catch
}//end of processMessage() method


/**
* This part of the probing is left as future work. Any algorithm regarding
* for DELAY parameter probing can be implemented in this method, as it
* is done for the throughput probing using modified packet pair algorithm.
*/
public void handleDelayProbing(){
        //future work
}


/**
* This part of the probing is left as future work. Any algorithm regarding
* for LOSS RATE parameter probing can be implemented in this method, as it
* is done for the throughput probing using modified packet pair algorithm.
*/
public void handleLossRateProbing(){
        //future work
}

/**
 * This method implements the THROUGHPUT probing. The control executive
 * will register this resident agent as a listener to the specified NIC, so it
 * can monitor for probing packets.
 * @param probeIdentification This is assigned by the server passed to this
 *      resident agent. It is used to check for the signature as well as to construct
 *      the probing result message format once it is done with collecting and
 *      computing the statistics
 * @param v6header This is the copy of the IPv6Header of the probing packets.
 * @param measurementInt This will define for how long this resident agent
 *      has to monitor for probing packets, since we don't want it to monitor
 *      for indefinite time period.
 * @param intInst This will define which interface instance to monitor for
 *      probing packets
 */
public void handleThroughputProbing(int probeIdentification, IPv6Header
                v6header, short measurementInt, byte intInst ){
        signature = probeIdentification;
        // add listener for the NIC that has been assigned
        // any packets arriving to this interface will be notified by the
        // control executive
        channel_ID =
                ProtocolStackEvent.getFromNICToInterfaceChannel(intInst);
        try{
                controlExec.addListenerToChannel(this, channel_ID);
                gui.sendText("Listening enabled on channel: "+channel_ID);
                gui.sendText("Now, I am monitoring probing packets, will
                        look for the signature ");
        }catch(ChannelException ce){
                gui.sendText(ce.toString());
        }//try-catch
}// end of handleThroughputProbing() method


/**
* This part of the probing is left as future work. Any algorithm regarding
* for UTILIZATION parameter probing can be implemented in this method, as it
* is done for the throughput probing using modified packet pair algorithm.
*/
public void handleUtilizationProbing(){
        //future work
}

/**
* This method will simply return the agent types in string format.
* @return The string representation of the agent types
*/
public String[] getAgentTypes(){
        return agentTypes;
}//getAgentTypes()

/**
* When the ControlExecutive receives an agent that is to replace
* an existing agent, the ControlExecutive calls the replaceAgent
* method on each of the ResidentAgentCustomers of that ResidentAgent.
* @param replacement The new ResidentAgent.
*/
public synchronized void replaceAgent( ResidentAgent agent){
}//replaceAgent()


/**
* When an agent is being replaced, the ControlExecutive will remove
* the old agent from all channels it is allowed to talk on and from
* all channels it is monitoring.  The uninstall method is called by
* the ControlExecutive upon replacement in order to allow the old
* agent a chance to perform any other cleanup that might be necessary,
* such as disposing of a user interface, for instance.
*/
public void uninstall(){ }


/**
* When an agent is about to be replaced, the ControlExecutive calls
* the transferState method, passing the old agent a copy of the new
* agent. The old agent should then call the receiveState method on
* the new agent, and pass to the new agent any messages that should
* be passed.
* @param replacement The agent that is replacing the old agent.
*/
public void transferState( ResidentAgent replacement ){ }


/**
* This method is called one or more times by an agent that is about
* to be replaced by this agent. The purpose of this method is to
* enable a state transfer from the old agent to the new agent.
* @param message The message to be passed from the old agent to the
*       new agent.
*/
public void receiveState(Message message){ }


/**
* When a ResidentAgent requests a flow by calling the requestFlow method
* of the ControlExecutive, the agent expects to be assigned a flow and
* to be notified when that flow is assigned. This method provides a
* mechanism for notifying the ResidentAgent that a FlowResponse has
* arrived.
*/
public void receiveFlowResponse(FlowResponse flowResponse){ }


/**
* Some resident agents are accessed by Objects on the router. This
* method provides the means for communication between an Object on
* the router and this ResidentAgent.
* @param message The Message the Object sends to this ResidentAgent.
* @return The Message this ResidentAgent sends back to the Object
*        performing the query.
*/
public Message query( Message message ){ return message; }


/**
* Returns an array that contains the class names of the resident
* agents that this Object desires to be a ResidentAgentCustomer of.
* @return An array that contains the class names of the resident
* agents that this Object desires to be a ResidentAgentCustomer of.
*/
public String[] getMessageTypes(){ return messageTypes;}
```

```
/**
 * This method implements the SaamListener class. Since this resident agent
 * was registered as a listener for the specified interface instance (NIC),
 * it will receive a copy of every packet arriving to this interface.
 * Once it receives a copy of the arriving packet, it will handle according
 * the probing type. If the initiated probing process is throughput which
 * is done using modified packet pair algorithm, it will check the last
 * four bytes looking if it matches with the signature (the signature is
 * the same with the probe ID). If it does match, it will record the arriving
 * time of each packet. After it receives the second probing packet, the control
 * executive will de-register from listening to the NIC. The resident agent will
 * compute the probing parameter, and form the probing result message format
 * and send it back to the server. For throughput measurement, it will compute
 * the throughput using packet pair formula, and form the probe result, and
 * finally send it to the server.
 */
public void receiveEvent(SaamEvent se){
        switch( nna.getTypeOfProbingField() ){
                case NextNodeAct.DELAY :
                                        //future work
                 break;
                case NextNodeAct.LOSS_RATE:
                                        //future work
                 break;
                case NextNodeAct.THROUGHPUT:
                        ProtocolStackEvent pse = (ProtocolStackEvent)se;
                        int channel = pse.getChannel_ID();
                        byte[] packet = pse.getPacket();
                        byte[] lastFourBytes;

                        /double check if it comes frome the correct channel
                        if(channel==channel_ID){
                                IPv6Packet v6Packet = null;
                                try{
                                        v6Packet = new IPv6Packet(packet);
                                }catch( UnknownHostException uhe){
                                gui.sendText("Couldn't instantiate IPv6Packet ");
                                }//end try-catch
                        // Display the packet information of the received copy
                                gui.sendText("Received a copy of a
                                        Packet from channel: "+channel);
                                gui.sendText("Source: " +
                                v6Packet.getHeader().getSource().toString());
                                gui.sendText("Came from: "+ pse.getSource());
                                gui.sendText("Packet length = " + packet.length +
                                "; Payload length = " +
                                v6Packet.getPayload().length +
                                   "; Flow label = " +
                                v6Packet.getHeader().getFlowLabel());

        gui.sendText("Now checking for the signature... ");
                        // extract the last four bytes of the received copy
        lastFourBytes = Array.getSubArray(packet,
                packet.length -4, packet.length);
                //compare the last four bytes with the signature
if(PrimitiveConversions.getInt(lastFourBytes)== signature ){
        gui.sendText("This packet MATCHES with the signature ");
        / get the timestamp for this matching packet
        timestamps[pointer]=System.currentTimeMillis();
        gui.sendText("The timestamp is " + timestamps[pointer]);
        //the pointer keeps track of how many packets have arrived
        if (pointer++ == 1){
                //compute the packet seperation in terms of time
                long timestampseperation = timestamps[1]-timestamps[0];
                // compute the throughput with the packet pair algorith
                // formula shown above
                bandwidth = (int)( (packet.length*8)/
                        ((double)timestampseperation/1000.0d)  );
                gui.sendText("Packet pair algorithm measurement
                                is done ");
                gui.sendText("The packet pair seperation is " +
                        timestampseperation + " milliseconds");
                gui.sendText("The measured bandwidth is " +
                 (short) (bandwidth/1000) + " kilo bits per second" );
                pr.setPacketFormat( signature, (byte) 1,
                         NextNodeAct.THROUGHPUT, (short)
                        (bandwidth/1000) );
                gui.sendText( "Probe Result Info " + pr.toString() );
                gui.sendText( "The length of the message is " +
                                pr.length() );
                gui.sendText( "I am sending the result to
                                control executive " );
                controlExec.sendProbeResult(pr);
                controlExec.removeListenerFromChannel( this,
                                channel_ID );
                gui.sendText("Listening disabled on channel " + channel );
                 gui.sendText("End of Monitoring packets ");

        }//end if
}else{
        gui.sendText("This packet DOES NOT MATCH with signature
                in the last four bytes, waiting for another packet...");
            }//end if-else
        }//end if
        break;
        case NextNodeAct.UTILIZATION:
            //future work
        break;
        default:
            gui.sendText("ERROR: Unrecognized Probing Type");
    }//end switch-case
  }// end receiveEvent() method
}//end of class file
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX K. PACKETFACTORY CLASS SOURCE CODE

```
// Mar 2000[altinkaya] – modified to handle server probing
// Feb 2000[akkoc]   - modified
// 01Aug99 [Vrable]  - Created
package saam.control;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.Hashtable;
import java.util.Enumeration;
import java.util.Vector;
import java.util.TooManyListenersException;
import java.util.StringTokenizer;
import java.lang.reflect.Constructor;
import java.net.UnknownHostException;

import saam.net.*;
import saam.event.*;
import saam.message.*;
import saam.util.*;
import saam.residentagent.*;

/**
* A PacketFactory can be used to build SaamPackets for sending or
* to receive SaamPackets and extract their atomic elements.  These
* atomic elements are currently one of two types:  A subclass of
* saam.residentagent.ResidentAgent or a subclass of
* saam.message.Message.<p>
* A sender would instantiate a PacketFactory to build
* Saam Packets.  The PacketFactory's append methods receive
* Message Objects, ResidentAgent Objects, or a String that represents
* the class name of a ResidentAgent as parameters and then dynamically
* construct the appropriate header based on the number of elements
* received and the current time.  The getBytes method is used to
* retrieve the byte array that represents the SAAMPacket that has been
* constructed by this PacketFactory.<p>
* The ControlExecutive uses the PacketFactory to receive and parse
* SaamPackets.
*/
public class PacketFactory extends Thread
  implements SaamTalker, SaamListener{

  private final boolean guiActive = true;
private SAAMRouterGui gui;
private ControlExecutive controlExec;
private boolean started = false;
private boolean firstEvent = true;
private boolean bytesRetrieved;
private byte[] packet,DCMpacket,PNpacket,UCMpacket;
private byte numberOfMessages;
private Loader loader;
private Class message;
private SaamEvent currentEvent;
private Thread owner;
//private static int instanceNumber;
private Object theLock = new Object();

// xie
private FIFOQueue inputQueue = new FIFOQueue(1000);

/**
 * Use the no-args constructor to begin constructing packets
 * on the sending side.
 */
//no-args constructor doesn't come for free when we have
//another constructor
public PacketFactory(){
 //instanceNumber++;
 //gui = new SAAMRouterGui(toString() + "("+ instanceNumber+")");
 gui = new SAAMRouterGui("Output "+ toString());
 gui.setTextField("I construct outbound packets");
}

/**
 * This constructor is not available to Objects outside the
 * saam.control package.  The ControlExecutive uses this constructor
 * to receive and parse SAAMPackets.  The PacketFactory passes the
 * atomic elements (either ResidentAgents or Messages) up to the
 * ControlExecutive for further processing.
 * @param controlExec The ControlExecutive that is to receive
 * updates from this PacketFactory.
 */
PacketFactory(ControlExecutive controlExec){
 //this();
 gui = new SAAMRouterGui("Input " + toString());
 gui.setTextField("I Listen for inbound packets");
 this.controlExec=controlExec;
 loader = new Loader();
```

```
    //*****************************
    //**Listen to desired Channels**
    //*****************************
    int channel_ID =
     ProtocolStackEvent.PACKETFACTORY_CHANNEL;
    try{
     controlExec.addListenerToChannel(this, channel_ID);
     gui.sendText("Listening to channel: "+channel_ID);
    }catch(ChannelException ce){
     gui.sendText(ce.toString());
    }//try-catch

    //***************************************
    //**Register to talk on desired Channels**
    //***************************************
    channel_ID = ControlExecutive.SAAM_CONTROL_PORT;
    try{
      controlExec.addTalkerToChannel(this,
        channel_ID);
      gui.sendText("Talking enabled on channel: " + channel_ID);
    }catch(ChannelException ce){
      gui.sendText(ce.toString());
    }
channel_ID =
ProtocolStackEvent.FROM_PACKETFACTORY_TO_SERVERAGENT_CHANNEL;
   try{
     controlExec.addTalkerToChannel(this, channel_ID);
     gui.sendText("Talking enabled on channel: " + channel_ID);
   }catch(ChannelException ce){
    gui.sendText(ce.toString());
   }
start();
 }

 /**
  * When instantiated to receive packets, the PacketFactory
  * Thread waits until a SAAMPacket arrives, then it calls
  * the processPacket method.
  */
 public void run(){
   while(true){
     gui.sendText("\n Inside PacketFactory run()");
     synchronized (theLock){
       if (inputQueue.isEmpty()){
         started = false;
```

```
          try{
            gui.sendText("Waiting...");
            theLock.wait();
            gui.sendText("Continuing");
          }
          catch(InterruptedException e){
            gui.sendText("Interrupted exception catched");
          }
        }// end if

        packet = (byte[]) inputQueue.dequeue();

      }// end synchronization

      processPacket();

    }//while(true)
}

/**
 * This method is called by the Channels this Object has registered to
 * monitor when a talker sends events on those Channels.
 * @param se The SaamEvent to be communicated.

public synchronized void receiveEvent(SaamEvent se){
*/
public  void receiveEvent(SaamEvent se){

  gui.sendText("\n---- Got a packet");
  currentEvent = se;
  ProtocolStackEvent psec = (ProtocolStackEvent)currentEvent;

  byte[] newcomer = psec.getPacket();
  gui.sendText("\n New packet has length = " + newcomer.length);

  synchronized(theLock){
    inputQueue.enqueue((Object) newcomer);
    if (!started){
      started = true;
      gui.sendText("\n Waking up the processPacket thread");
      theLock.notify();
    }// end if
  }
}

/**
```

```
 * This method is used to extract the individual Class
 * Objects that are represented in the packet.  These Class
 * Objects are either of type 0 (ResidentAgent) or 1 (Message).<p>
 * If a ResidentAgent is received, a Class Object is created
 * that represents the agent.  That Class Object is then sent to
 * the ControlExecutive for screening and agent instantiation.<p>
 * If a Message is received, that Message is instantiated and sent
 * to the ControlExecutive for further processing.
 */

private void processPacket() {
 int channel = currentEvent.getChannel_ID();
 String eventSource = (String)currentEvent.getSource();

 //see saam.util for PrimitiveConversions and Array classes
 long timeStamp = PrimitiveConversions.getLong(
   Array.getSubArray(packet,0,8));
 numberOfMessages=packet[8];
 gui.sendText("packet arrived: " +
  "\n  source:      " + eventSource +
  "\n  channel:     " + channel +
  "\n  size:        " + packet.length +
  "\n  # of Messages: " + numberOfMessages +
  "\n  timeStamp:   " + timeStamp);

 //now we trim the packet by removing the header.
 packet = Array.getSubArray(packet,9,packet.length);
 byte type = packet[0];
            gui.sendText("packet type ..." + type );


 //used to track the current position in the array.
 int index = 0;

           for(int i=1;i<=numberOfMessages;i++){
     gui.sendText("\nProcessing Element["+i+"]:");
//   int index = 0;
            type = packet[index++];
            gui.sendText("packet type ..." + type );

     switch(type){

   case 0:
   case 1:
           //extract and process each atomic element of the packet
           //separately.  Here we assume the packet is a properly
```

```
             //formatted SAAMPacket when it arrives, and that the
             //length is less than the max allowed.

          gui.sendText("  type:   "+type);
          //retrieve the number of bytes the class name occupies
          byte nameLength = packet[index++];

          //extract the name of the class file as a byte array
          byte[] elementNameArray = Array.getSubArray(
           packet,index, index+nameLength);
          index+=nameLength;

          //convert the name back into a String
          String elementName = new String(elementNameArray);
          gui.sendText("  Name: "+elementName);

          //retrieve the length of the Object
          short length = PrimitiveConversions.getShort(
           Array.getSubArray(packet,index,index+2));
          gui.sendText("  Length:      "+length);
          index+=2;

          //retrieve the bytecode of the Object
          byte[] bytes = Array.getSubArray(
           packet,index,index+length);
          index+=length;

                         if (type == 0){
gui.sendText("This is a ResidentAgent");
      //Assume this class is of type ResidentAgent
      try{
       //Attempt to define the class using the current
       //class loader.
       loader.defClass(elementName, bytes);
      }catch(LinkageError le){
       //If the loader already has a definition for the class
       //a LinkageError will be thrown.  If this happens, we
       //need to instantiate a new class loader and use it to
       //define the class.  A nice little trick we learned from
       //page 55 of Jason Hunter's "Java Servlet Programming" book.
       gui.sendText(le.toString());
       gui.sendText("Class was previously loaded...");
       gui.sendText("Replacing old ClassLoader...");
       Loader newLoader = new Loader();
       newLoader.defClass(elementName, bytes);
      }
```

```
    try{
     //message is of type Class.
     message = Class.forName(elementName, true, loader);
    }catch(ClassNotFoundException cnfe){
     gui.sendText(cnfe.toString());
    }
    gui.sendText(message.toString());
    ResidentAgentEvent rae = new ResidentAgentEvent(
     eventSource,
     this,
     ControlExecutive.SAAM_CONTROL_PORT,
     message);
    try{
     gui.sendText("Forwarding on channel "+
      ControlExecutive.SAAM_CONTROL_PORT);
     controlExec.talk(rae);
    }catch(ChannelException tde){
     gui.sendText(tde.toString());
    }
                         }else {
    gui.sendText("This is a Message");
    //Assume this class is of type Message.
    try{
     //message is of type Class.
message = Class.forName(elementName);
    }catch(ClassNotFoundException cnfe){
System.out.println("error is here  ");
     {gui.sendText("Bytecode for: "+elementName+
      " not found.");
     }
    }

    try{
     //Call the constructor from within this Class that
     //takes a byte array as its only argument
     Constructor cons = message.getConstructor(
      new Class[] {byte[].class});

     //Create the instance of this Message
     Message instance = (Message)cons.newInstance(
       new Object[] {bytes});
     gui.sendText(instance.toString());
     MessageEvent me = new MessageEvent(eventSource, this,
      ControlExecutive.SAAM_CONTROL_PORT,  instance);
    //send this MessageEvent on the Control port.
    try{
```

```
             gui.sendText("Forwarding on channel "+
              ControlExecutive.SAAM_CONTROL_PORT);
             controlExec.talk(me);
            }catch(ChannelException tde){
        gui.sendText("problem occured here ");
             gui.sendText(tde.toString());
            }
           }catch(Exception e){
            //need to notify sender that we have no classfile
            //with this name
            gui.sendText(e.toString());
           }//try-catch
                              }
     break;

    //THIS CASE PROCESSES THE HEARBEATQUERY MESSAGE
    case 2:

     //retrieve the bytecode of the Object
     bytes = Array.getSubArray(packet,1,packet.length);
     gui.sendText("This is a  HeartbeatQuery message");

     //Create the instance of this Message
     HeartbeatQuery hbq = new HeartbeatQuery(bytes);
     gui.sendText(hbq.toString());

     MessageEvent hbqMe = new
MessageEvent(eventSource,this,ControlExecutive.SAAM_CONTROL_PORT,hbq);
     //send this MessageEvent on the Control port.
     try{
      gui.sendText("Forwarding on channel "+
      ControlExecutive.SAAM_CONTROL_PORT);
      controlExec.talk(hbqMe);
     }catch(ChannelException tde){
      gui.sendText(tde.toString());
     }

     break;

    //THIS CASE PROCESSES THE HEARBEATRESPONSE MESSAGE
    case 3:

     //retrieve the bytecode of the Object
     bytes = Array.getSubArray(packet,1,packet.length);
     gui.sendText("This is a  HeartbeatResponse message");
```

```
        //Create the instance of this Message
        HeartbeatResponse hbr = new HeartbeatResponse(bytes);
        gui.sendText(hbr.toString());

        MessageEvent hbrMe = new
MessageEvent(eventSource,this,ControlExecutive.SAAM_CONTROL_PORT,hbr);
        //send this MessageEvent on the Control port.
        try{
          gui.sendText("Forwarding on channel "+
          ControlExecutive.SAAM_CONTROL_PORT);
          controlExec.talk(hbrMe);
        }catch(ChannelException tde){
          gui.sendText(tde.toString());
        }


        break;

        case 4:
          gui.sendText("This is a DCM  Message");

          try{
            DCM  dcm = new DCM(packet);
            gui.sendText(dcm.toString());

            MessageEvent me = new MessageEvent( eventSource, this,
                                ControlExecutive.SAAM_CONTROL_PORT,dcm);
            //send this MessageEvent on the Control port.
            try{

              gui.sendText("Forwarding on channel "+

ProtocolStackEvent.FROM_PACKETFACTORY_TO_SERVERAGENT_CHANNEL);
                              controlExec.talk(pse);
          gui.sendText("Forwarding on channel "+
                     ControlExecutive.SAAM_CONTROL_PORT);
                     controlExec.talk(me);

            }catch(ChannelException tde){
              gui.sendText(tde.toString());
              tde.printStackTrace();
            }
          }catch(Exception e){
            gui.sendText(e.toString());
            e.printStackTrace();
        }//try-catch
```

```
   break;

   case 5:
     gui.sendText("This is a UCM  Message");
     try{
       UCM ucm  =  new UCM(packet);
       gui.sendText("\n"+ucm.toString());
       MessageEvent me = new MessageEvent( eventSource, this,
             ControlExecutive.SAAM_CONTROL_PORT,ucm);
       //send this MessageEvent on the Control port.
       try{
         gui.sendText("Forwarding on channel "+
            ControlExecutive.SAAM_CONTROL_PORT);
         controlExec.talk(me);
       }catch(ChannelException tde){
         gui.sendText(tde.toString());
       }
     }catch(Exception e){
       gui.sendText(e.toString());
   }//try-catch
   break;

   case 6:
     gui.sendText("This is a ParentNotification Message");
     //Assume this class is of type Message.
     try{
       ParentNotification pn = new ParentNotification(packet);
       MessageEvent me = new MessageEvent( eventSource, this,
                                  ControlExecutive.SAAM_CONTROL_PORT,pn);
       //send this MessageEvent on the Control port.
       try{
         gui.sendText("Forwarding on channel "+
         ControlExecutive.SAAM_CONTROL_PORT);
         controlExec.talk(me);
       }catch(ChannelException tde){
         gui.sendText(tde.toString());
       }
     }catch(Exception e){
       gui.sendText(e.toString());
   }//try-catch
   break;

// this case statement is added by altinkaya in March 2000
// as a requirement for the server probing process
case Message.PREVIOUS_NODE_ACT:
    gui.sendText("This is a Previous Node Activation Message");
```

```
          PreviousNodeAct pna = new PreviousNodeAct();
          byte typeOfProbing;
          int probeID;
          IPv6Header v6Header=null;
          short payloadLength;

          typeOfProbing = packet[index++];
          probeID = PrimitiveConversions.getInt(Array.getSubArray(packet, index,
   index=index+4));

         try{
          v6Header = new IPv6Header( Array.getSubArray(packet, index, index = index +
   40 ));
         }catch(UnknownHostException uhe){
          gui.sendText("An exception occured while trying to read ipv6Header ");
         }//end try-catch
         payloadLength = PrimitiveConversions.getShort( Array.getSubArray(packet,
   index, index = index + 2 ));

         pna.setPacketFormat( typeOfProbing, probeID, v6Header, payloadLength );

         gui.sendText("Type of probing " + pna.getTypeOfProbing( typeOfProbing ) );
         gui.sendText("Probing idendification " + probeID );

         MessageEvent me = new MessageEvent( eventSource, this,
   ControlExecutive.SAAM_CONTROL_PORT, pna );

         try{
          gui.sendText( "Forwarding on channel " +
   ControlExecutive.SAAM_CONTROL_PORT);
          controlExec.talk(me);
         }catch( Exception e){
          gui.sendText( e.toString());
         }
          break;

         // this case statement is added by altinkaya in March 2000
         // as a requirement for the server probing process
        case Message.NEXT_NODE_ACT:
             gui.sendText("This is a Next Node Activation Message");

         NextNodeAct nna = new NextNodeAct();
         byte typeOfProbingNNA;
         int probeIDNNA;
         IPv6Header v6HeaderNNA=null;
```

```
          short measurementIntervalNNA;
          byte nicID;


          typeOfProbingNNA = packet[index++];
        probeIDNNA = PrimitiveConversions.getInt(Array.getSubArray(packet, index,
index=index+4));

        try{
          v6HeaderNNA = new IPv6Header( Array.getSubArray(packet, index, index =
index + 40 ));
        }catch(UnknownHostException uhe){
          gui.sendText("An exception occured while trying to read ipv6Header ");
        }//end try-catch
        measurementIntervalNNA = PrimitiveConversions.getShort(
Array.getSubArray(packet, index, index = index + 2 ));
        nicID = packet[index++];

        nna.setPacketFormat( typeOfProbingNNA, probeIDNNA, v6HeaderNNA,
measurementIntervalNNA, nicID );

        gui.sendText("Type of probing " + nna.getTypeOfProbing( typeOfProbingNNA ) );
        gui.sendText("Probing idendification " + probeIDNNA );

        MessageEvent meNNA = new MessageEvent( eventSource, this,
ControlExecutive.SAAM_CONTROL_PORT, nna );

        try{
          gui.sendText( "Forwarding on channel " +
ControlExecutive.SAAM_CONTROL_PORT);
          controlExec.talk(meNNA);
        }catch( Exception e){
          gui.sendText( e.toString());
        }//end try-catch
       break;


       // this case statement is added by altinkaya in March 2000
       // as a requirement for the server probing process
       case Message.PROBE_RESULT:
          gui.sendText("This is a Probe Result Message ");

          ProbeResult pr = new ProbeResult();
          int probeIDPR;
          byte numberResultsPR;
          byte typeofResultPR;
          short measurementResultPR;

        probeIDPR = PrimitiveConversions.getInt(Array.getSubArray(packet, index,
index=index+4));
        numberResultsPR = packet[index++];
        typeofResultPR = packet[index++];
        measurementResultPR = PrimitiveConversions.getShort(
Array.getSubArray(packet, index, index = index + 2 ));
        pr.setPacketFormat( probeIDPR, numberResultsPR, typeofResultPR,
measurementResultPR );
        gui.sendText("Information about the Probe Result " + pr.toString() );
        MessageEvent mePR = new MessageEvent( eventSource, this,
ControlExecutive.SAAM_CONTROL_PORT, pr );

        try{
         gui.sendText( "Forwarding on channel " +
ControlExecutive.SAAM_CONTROL_PORT);
         controlExec.talk(mePR);
        }catch( Exception e){
         gui.sendText( e.toString());
        }//end try-catch
         break;

       default:
        gui.sendText("Packet type unrecognized: "+type);
         //packet type is unrecognized.  Here we could
         //extract a channel_ID that could be embedded
         //in the packet, and then send the unrecognized
         //element on that channel.
     }//end switch
   }//for
  }//processPacket()

  /**
   * This method can be used to append a Message to an outgoing
   * SAAMPacket.  To later retrieve the entire packet (with header)
   * as a byte array, call the getBytes method.
   * @param me The Message to be appended.
   */
  public void append(Message me){

/*
   if(bytesRetrieved){
     packet=null;
     numberOfMessages=0;
     bytesRetrieved = false;
```

```
    }
    byte type = me.getType();


     String name = me.getClass().getName();
     gui.sendText("appending "+name);
     byte nameLength = (byte)name.getBytes().length;
     byte[] parameters = me.getBytes();

    //here we could check the length of the parameter array supplied
    //with the length returned from the length() method call.
     short paramLength = (short)parameters.length;
     //now append the Message to the packet byte array
     packet = Array.concat(packet,type);
     packet = Array.concat(packet,nameLength);
     packet = Array.concat(packet,name.getBytes());
     packet = Array.concat(packet,PrimitiveConversions.getBytes(paramLength));
     packet = Array.concat(packet,parameters);
     //increment the count of messages in this packet
      numberOfMessages++;

     gui.sendText("Appended Message:" +
     "\n  Type:          " + type +
     "\n  name:          " + name +
     "\n  param length:  " + paramLength +
     "\n  # of elements: " + numberOfMessages +
     "\n  packet length: " + packet.length+"\n");

*/

    if(bytesRetrieved){
     packet=null;
     numberOfMessages=0;
     bytesRetrieved = false;
    }
    byte type = me.getType();
    byte[] parameters = me.getBytes();
    //here we could check the length of the parameter array supplied
    //with the length returned from the length() method call.
    short paramLength = (short)parameters.length;


    //IN THIS SWITCH STATEMENT CASE 1 IS DESIGNED TO SUPPORT THE
    //OLD VERSION MESSAGE TYPE
    //
    //CASE 1 :  FOR OLD MESSAGE TYPE(TYPE=1)
```

```
//CASE 2 :  FOR THE HEARTBEATQUERY TYPE(TYPE=2)
//CASE 3 :  FOR THE HEARTBEATRESPONSE TYPE(TYPE=3)
//
//OTHER CASES WILL BE IMPLEMENTED FOR OTHER MESSAGE TYPES
switch(type){

 case 1://FOR OLD VERSION MESSAGE TYPE

  String name = me.getClass().getName();
  byte nameLength = (byte)name.getBytes().length;


  //now append the Message to the packet byte array
  packet = Array.concat(packet,type);
  packet = Array.concat(packet,nameLength);
  packet = Array.concat(packet,name.getBytes());
  packet = Array.concat(packet,
   PrimitiveConversions.getBytes(paramLength));
  packet = Array.concat(packet,parameters);

  gui.sendText("Appended Message:" +
   "\n  Type:          " + type +
   "\n  name:          " + name +
   "\n  param length:  " + paramLength +
   "\n  # of elements: " + numberOfMessages +
   "\n  packet length: " + packet.length+"\n");

  break;


 case 2://FOR THE HEARTBEATQUERY MESSAGE

  packet = Array.concat(packet,type);
  packet = Array.concat(packet,parameters);
  gui.sendText("Appended Message:" +
   "\n  Type:          " + type +
   "\n  name:          " + "HeartbeatQuery" +
   "\n  param length:  " + parameters.length +
   "\n  packet length: " + packet.length+"\n");

  break;

 case 3://FOR THE HEARTBEATRESPONSE MESSAGE

  packet = Array.concat(packet,type);
  packet = Array.concat(packet,parameters);
```

```
     gui.sendText("Appended Message:" +
      "\n  Type:          " + type +
      "\n  name:          " + "HeartbeatResponse" +
      "\n  param length:  " + parameters.length +
      "\n  packet length: " + packet.length+"\n");
     gui.sendText(((HeartbeatResponse)(me)).toString());
     break;

  // this case statement is added by altinkaya in March 2000
  // as a requirement for the server probing process
              case Message.PREVIOUS_NODE_ACT://for the Previuos Node
Activation Message
     packet = Array.concat(packet,parameters);
     gui.sendText("Appended Message:" +
      "\n  Type:          " + type +
      "\n  name:          " + "Previuous Node Activation" +
      "\n  param length:  " + parameters.length +
      "\n  payload length: " + packet.length+"\n");
     gui.sendText(((PreviousNodeAct)(me)).toString());
     break;

      // this case statement is added by altinkaya in March 2000
  // as a requirement for the server probing process
      case Message.NEXT_NODE_ACT://for the Next Node Activation Message
     packet = Array.concat(packet,parameters);
     gui.sendText("Appended Message:" +
      "\n  Type:          " + type +
      "\n  name:          " + "Next Node Activation" +
      "\n  param length:  " + parameters.length +
      "\n  payload length: " + packet.length+"\n");
     gui.sendText(((NextNodeAct)(me)).toString());
     break;

              // this case statement is added by altinkaya in March 2000
  // as a requirement for the server probing process
              case Message.PROBE_RESULT://for the Probe Results
     packet = Array.concat(packet,parameters);
     gui.sendText("Appended Message:" +
      "\n  Type:          " + type +
      "\n  name:          " + "Probe Result" +
      "\n  param length:  " + parameters.length +
      "\n  payload length: " + packet.length+"\n");
     gui.sendText(((ProbeResult)(me)).toString());
     break;
     default:

       gui.sendText("Packet type unrecognized: "+type);

    }//end switch

   //increment the count of messages in this packet
   numberOfMessages++;

  } //end of append


  //this method is added by mustafa Feb 20,2000
  public void append( byte [] packetArray ){

   if(bytesRetrieved){
     packet=null;
     numberOfMessages=0;
     bytesRetrieved = false;
   }
  packet = Array.concat(packet, packetArray);
  numberOfMessages++;

  }

  /**
   * This method can be used to append a DCM message to an outgoing
   * SAAMPacket.  To later retrieve the entire packet (with header)
   * as a byte array, call the getDCMBytes method.
   * @param downWard The DCM message to be appended.
   */

 public void appendDCM( DCM downWard){
 DCMpacket = null;
   gui.sendText(" Appending a dcm message before sending downward with lengh");
   DCMpacket = Array.concat(DCMpacket,downWard.getBytes());
   gui.sendText(" Appending a dcm message before sending downward with lenght
"+DCMpacket.length);
  }//end of appendDCm

 /**
  * This method can be used to append a PN message to an outgoing
  * SAAMPacket.  To later retrieve the entire packet (with header)
  * as a byte array, call the getPNBytes method.
  * @param downWard The PN message to be appended.
 */
public void appendPN( ParentNotification pn){
PNpacket =null;
  gui.sendText(" Appending a PN message before sending downward with lengh");
  PNpacket = Array.concat(PNpacket,pn.getBytes());
  gui.sendText("after appending PN is "+PNpacket.length);

}//end of appendDCm

/**
 * This method can be used to append a PN message to an outgoing
 * SAAMPacket.  To later retrieve the entire packet (with header)
 * as a byte array, call the getPNBytes method.
 * @param downWard The PN message to be appended.
 */
public void appendUCM( UCM upWard){
  UCMpacket =null;
  gui.sendText(" Appending a UCM message before sending upward");
  UCMpacket = Array.concat(UCMpacket,upWard.getBytes());
}//end of appendUCM

/**
 * This method can be used to append a ResidentAgent to an outgoing
 * SAAMPacket.  To later retrieve the entire packet (with header)
 * as a byte array, call the getBytes method.
 * @param ra The ResidentAgent to be appended.
 */
public void append(ResidentAgent ra) throws IOException{
  String name = ra.getClass().getName();
  append(name);
}
/**
 * This method can be used to append a ResidentAgent by name to an
 * outgoing SAAMPacket.  To later retrieve the entire packet
 * (with header) as a byte array, call the getBytes method.
 * @param residentAgentClassName The String name of the ResidentAgent
 * classfile to be appended.
 */
public void append(String residentAgentClassName)
  throws IOException{
  if(bytesRetrieved){
    packet=null;
    numberOfMessages=0;
    bytesRetrieved = false;
  }
  byte type = 0;
```

```
   String name = residentAgentClassName;
// String fileName =
"C:\\WINNT\\Profiles\\administrator\\Desktop\\Java\\saamjuly\\saamxpand1"+File.separ
atorChar +
   String fileName = "C:\\saamMustafa"+File.separatorChar +
    residentAgentClassName.replace('.',File.separatorChar);
   fileName+=".class";
   gui.sendText("File name: "+fileName);
   FileInputStream fis = null;
   try{
    fis = new FileInputStream(fileName);
   }catch(IOException ioe){
    throw new IOException(
      "Problem reading ResidentAgent: "+fileName);
   }
   byte nameLength = (byte)name.getBytes().length;
   byte[] byteCode = new byte[fis.available()];
   short length = (short)fis.read(byteCode);

   packet = Array.concat(packet,type);
   packet = Array.concat(packet,nameLength);
   packet = Array.concat(packet,name.getBytes());
   packet = Array.concat(packet,
   PrimitiveConversions.getBytes(length));
   packet = Array.concat(packet,byteCode);
   numberOfMessages++;

   gui.sendText("Appended ResidentAgent:" +
     "\n  Type:              " + type +
     "\n  name:              " + name +
     "\n  byteCode length:   " + length +
     "\n  # of elements:     " + numberOfMessages +
     "\n  packet length:     " + packet.length+"\n");

 }

 /**
  * Appends a header to the byte array.  The header conforms
  * to the structure of a SAAMHeader.
  */
 private void appendHeader(){
  byte[] timeStamp = PrimitiveConversions.getBytes(
   System.currentTimeMillis());
   packet = Array.concat(numberOfMessages,packet);
   packet = Array.concat(timeStamp,packet);
   gui.sendText("Appended header:"+
      "\n  timeStamp:    "+PrimitiveConversions.getLong(
        Array.getSubArray(packet,0,8))+
      "\n  # of updates: "+packet[8] +
      "\n  packet length: "+packet.length+"\n");
  }

  /**
   * Returns a byte array that conforms to the structure of
   * a SAAMPacket.
   * @return A byte array that conforms to the structure of
   * a SAAMPacket.
   */
  public byte[] getBytes(){
    appendHeader();
    bytesRetrieved = true;
    return packet;
  }

  /**
   * Returns a byte array that conforms to the structure of a DCMPacket.
   * @return A byte array that conforms to the structure of DCMPacket.
   */

   public byte[] getDCMBytes(){
    return DCMpacket;
  }

  /**
   * Returns a byte array that conforms to the structure of a PNPacket.
   * @return A byte array that conforms to the structure of PNPacket.
   */
  public byte[] getPNBytes(){
    return PNpacket;
  }

  /**
   * Returns a byte array that conforms to the structure of a UCMPacket.
   * @return A byte array that conforms to the structure of UCMPacket.
   */
  public byte[] getUCMBytes(){
    return UCMpacket;
  }

  /**
   * Returns the current length of the packet.
   * @return The current length of the packet.
  */
 public int length(){
   try{
     return packet.length;
   }catch(NullPointerException npe){
     return 0;
   }
 }

 /**
  * Returns a <code>String</code> representation of this object
  * @return The <code>String</code> representation of this object
  */
 public String toString(){
   return "Packet Factory";
 }

}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX L. SERVERAGENT CLASS SOURCE CODE

```
// Mar 2000 [altinkaya] – modified to handle server probing
package saam.residentagent.server;

import saam.control.*;
import saam.residentagent.*;

import saam.server.*;
import saam.event.*;
import saam.message.*;
import saam.util.*;
import saam.event.SaamListener;


public class ServerAgent implements ResidentAgent, MessageProcessor, SaamListener{

  private SAAMRouterGui gui;
  private ControlExecutive controlExec;
  private Server myServer;


  private String[] messageTypes =
    {"saam.message.Hello",
     "saam.message.FlowRequest",
     "saam.message.LinkStateAdvertisement",
     "saam.message.Configuration",
     "saam.message.HeartbeatQuery",
     "saam.message.HeartbeatResponse",
     "saam.message.ProbeResult"};



 public void install(ControlExecutive controlExec){
   gui=new SAAMRouterGui("ServerAgentSymetric");
   this.controlExec=controlExec;
   controlExec.registerMessageProcessor(this);
/*
   try{

   controlExec.addListenerToChannel(this,
ProtocolStackEvent.FROM_PACKETFACTORY_TO_SERVERAGENT_CHANNEL);
   gui.sendText("Registered for channel " +
ProtocolStackEvent.FROM_PACKETFACTORY_TO_SERVERAGENT_CHANNEL +
"");
```

```
    }
    catch(ChannelException ce){

     gui.sendText(ce.toString());
    }

    */
    myServer = new Server("classObject", controlExec);
//   myServer = new Server("database", controlExec);

    gui.sendText("\nCalling My Server method: autoconfig()");
    myServer.autoConfig();
  }

  public void processMessage(Message message){
   try{
    if(message instanceof Hello){
     gui.sendText("Received Message: "+((Hello)message));
     gui.sendText("Calling Server method: processHello()");
     myServer.processHello((Hello)message);
    }else if(message instanceof FlowRequest){
     FlowRequest request = (FlowRequest)message;
     gui.sendText("Received Message: "+ request);
     gui.sendText(
      "Calling Server method: processFlowRequest()");
     myServer.processFlowRequest((FlowRequest)message);

    }else if(message instanceof LinkStateAdvertisement){
     gui.sendText("Received Message: "+
      ((LinkStateAdvertisement)message));
     gui.sendText("Calling Server method: processLSA()");
     myServer.processLSA(
      (LinkStateAdvertisement)message);
    }
    //below adde by akkoc
    else if(message instanceof Configuration){
     gui.sendText("Received Message: "+
      ((Configuration)message));
     gui.sendText("Calling Server method: processConfiguration()");
     myServer.processConfiguration((Configuration)message);
    }

    //added efraim
    else if(message.getType() == (byte)2){
     gui.sendText("Received Message: "+ ((HeartbeatQuery)message));
     gui.sendText("Calling Server method: processHeartbeatQuery()");
```

```
      myServer.processHeartbeatQuery((HeartbeatQuery)message);
     }

    //added efraim
    else if(message.getType() == (byte)3){
      gui.sendText("Received Message: "+ ((HeartbeatResponse)message));
      gui.sendText("Calling Server method: processHeartbeatResponse()");
      myServer.processHeartbeatResponse((HeartbeatResponse)message);
    }

    //added altinkaya March 00
    else if(message.getType() == Message.PROBE_RESULT){
                                  gui.sendText("Received Message: " +
                                  ((ProbeResult)message) );
                                  gui.sendText("Calling Server method:
                                  processProbeResult() " );
                                  myServer.processProbeResult( (ProbeResult)message );
                         }
   }catch(Exception e){message = null;}
  }

  public String[] getMessageTypes(){
   gui.sendText("Server queried my message types");
  //  gui.sendText("Sending: "+messageTypes[0]);
   return messageTypes;
  }
  public String toString() {
   String it = "ServerAgentSymetric listening for: ";
   for(int i=0;i<messageTypes.length;i++){
    it += "\n" + messageTypes[i];
   }
   return it;
  }//toString()

  //the following methods are stubbed out as they are not used.
  public void uninstall(){
  }
  public Message query(Message message){
   return message;
  }
  public void transferState(ResidentAgent replacement){
  }
  public void receiveState(Message message){
  }
  public void receiveFlowResponse(FlowResponse flowResponse){
  }
```

```
 public void receiveEvent(SaamEvent event){
 /*
     ProtocolStackEvent pse = (ProtocolStackEvent)event;
     DCM dcm = new DCM (pse.getPacket());
     gui.sendText("Received  DCM Message: ");
     gui.sendText("Calling Server method: processDCM()");
     myServer.processDCM(dcm);
 */

  }
}
```

APPENDIX M. SERVER CLASS SOURCE CODE

```
// Mar 2000 [altinkaya] – modified to handle server probing
// Feb 2000 [akkoc] - modified
//01august99[vrable] - created
package saam.server;

import saam.EmulationTable;
import saam.Translator;
import saam.*;

import saam.net.*;
import saam.message.*;
import saam.control.*;
import saam.event.*;
import saam.router.*;
import saam.util.*;
import java.net.*;
import java.util.*;
import java.io.*;


import java.awt.*;
import javax.swing.*;
import java.awt.event.ActionEvent;


/**
 * The <em>Server</em> is an object within the SAAM architecture that
 * maintains a picture of the network for use in assigning flows to paths.
 */

public class Server implements  Runnable{

 //declare class variables

 /** Contains what is known about the network. */
 private PathInformationBase PIB;

 /** Enables the Server to receive and send particular types of messages. */
 public ControlExecutive controlExec;
 /** A maximum number of hops that a search for different paths may take. */
 private int Hmax = 4;

 /**
 * Used to lookup what flow id should be used to send out control messages
 * to specified routers.
 */
private Hashtable flowLookUp = new Hashtable();

/** Used to assign the right number of service level pipes to interfaces in
 * this SAAM region. Only used during initialization -- later were assume
 * SLPs are known to routers
 */
private int numOfServiceLevels = 4;

/**
 * The value assigned to flow ids that can not be supported. This should be
 * switched over to 0 as soon as routers are converted.
 */
public static int FLOWNOTSUPPORTABLE = 99;

public static int INITIALDELAY = 0;
public static int INITIALLOSSRATE = 0;
public static int INITIALTHROUGHPUT = 10000;

public static int RETURNFLOWDELAY = 50;
public static int RETURNFLOWLOSSRATE = 50;
public static int RETURNFLOWTHROUGHPUT = 1000;

public static int ROUTERNOTINPIB = 0;

public static int NOSUPPORTABLEPATHINPIB = 0;

public static int SERVERNODEID = 1;

public static int FLOWTOSERVER = 0;

public static int PSUEDORANDOMSOURCEPORT = 8000;

public static int INITIALPATHID = 0;

public static int INITIALHEIGHTOFSEARCH = 1;
public static int INCREMENTATIONOFSEARCH = 1;
public static int DESTINATIONNODE = 0;

public static int INITIALZERO = 0;


/** Defines with the appropriate IPv6 address of this server. */
//private String serverIPv6 = controlExec.getServerIP().toString();
```

```
private String serverIPv6 = "99.99.99.99.1.0.0.0.0.0.0.0.0.0.0.2";

/** Time when the all possible paths were found. */
private long timeOfLastPIBBuild = System.currentTimeMillis();

/**
 * The amount of time that we want to have between rebuilding of paths. This
 * is not currently implemented.
 */
private long timeBetweenPIBBuilds = 120000; // 2 minutes (or 120 sec)

/** A boolean that will allow the showing of comments. */
private boolean showComments = true;

//private SAAMRouterGui gui;
public SAAMRouterGui gui;

// 2000 akkoc added
private int sequenceNumber = 1;
private static final int CTS = 0;    //SINCE ITS SERVER BY ITSELF
private int hopCount;


private static byte serverType;
private static int flowId;
private static byte metricType;
private static int cycleTime;
private static int globalTime;

//1 Feb 2000 akkoc added
private IPv6Address ServerId;

        //March 2000 ALTINKAYA added
private boolean probeMessagesSent = false;
private int counter = 1;
private PreviousNodeAct pna = new PreviousNodeAct();
private NextNodeAct nna = new NextNodeAct();

private int lastUsedFlow_ID = 0;
private short seqNumInit = 0;
HeartbeatResponse hbr = new HeartbeatResponse(seqNumInit,lastUsedFlow_ID);
HeartbeatController hbController;
private boolean isMain;
private boolean isMainDown = false;
private short lastSequenceNumber = 0;
private Vector recentMissedSequences = new Vector();
```

```
double tmax = 16.0d;
double tmin = 1.0d;
BannerFrame bf = new BannerFrame("");
long lastResponseTime = System.currentTimeMillis();
long lastQueryTime = System.currentTimeMillis();
long currentTime = System.currentTimeMillis();
int firstQueryTime = 90000;
long lastDCMTime = System.currentTimeMillis();


//normal failure detection
//private boolean testCase0 = true;


// TEST CASE #1
// this case tests the lost heartbeat response
// after the 10th heartbeat query message (if testCase1 is true):
// * 3th respone will not be send
// * 4th response will not be send
// then responses will be sent correctly
private boolean testCase1 = false;


// TEST CASE #2
// this case tests the sequence number functionality
// after the 10th heartbeat query message (if testCase2 is true):
// * 11th respone will not be send
// * 12th response will be send with sequence # 11
// then responses will be sent correctly
private boolean testCase2 = false;


// TEST CASE #3
// this case tests the usage of DCM as unsolicited heartbeat
// after the 10th heartbeat query message (if testCase3 is true):
// * 11th respone will not be send
// * 12th response will not be send
// * 13th respone will not be send
// * 14th response will not be send
// * 15th response will not be send
// then responses will be sent correctly
private boolean testCase3 = false;


/**
 * Constructs a server that will use a specified type of <em>Path Information
```

```
   * Base</em>. The PIB may be in the form of a database structure (which
   * requires an existing ODBC configured local database) or a class
   * object structure. The control executive is the interface to the IPv6
   * protocol stack, in order for messages to flow to and from the network.
   * The final step taken is the deletion of all existing data, which is
   * important only in a database structure since a class object structure is
   * volatile.
   * @param type The type of structure that the PIB is to assume.
   * @param controlExec The control executive that will exchange messages
   * with this server.
   */
  public Server(String type, ControlExecutive controlExec){
   if (type == "database")
    PIB = new DatabaseStructure();
   else
    PIB = new ClassObjectStructure();

   this.controlExec = controlExec;
   gui=new SAAMRouterGui("Server");


   // 1feb 2000 akkoc added
   ServerId = controlExec.getRouterId();

   PIB.deleteAllData();
  }

  public Server(){ }//temp for time measurement


//*************************************************************************
// These methods handle external network communications from routers
//************************************************************************/


  /**
   * Receives Hello messages from routers and then processes them.  It starts
   * building a vector of IPv6Addresses from the interfaces included in the
   * Hello message. This vector is passed to the PIB's doesRouterExist() which
   * determines if a router with any of these interfaces have been identified
   * before. If this is a new router, a new unique node id is assigned.<p>
   * For each of the interfaces identified in the Hello message, if this
   * interface was is not known to the PIB, check to see if the corresponding
   * link is known to the PIB. If this link is not known to the PIB, add it.
   * Next, add the new interface between the node and link. Also, add each
   * service level pipe that is assigned within this SAAM region.<p>
```

```
 * The next step is to rebuild the paths that are possible across the network
 * now considering this new hello message. The frequency of these rebuilds is
 * not a major concern in a controlled environment, but will need to be
 * addressed later. Finally, a flow request is create and received for
 * communicating back to this node. This is only possible if the PIB's
 * determineAllPossiblePaths() has been executed after the processing of this
 * particular hello message, if this a new router. After all paths to each
 * known router are found, we finish this method with a call to
 * determineEffectiveQoSForPaths(). The call to
 * determineEffectiveQoSForPaths() ensures that even if no QoS parameters are
 * known about these new parts of the network, that at least some initial
 * values will be assigned. This initialization allows the new paths to be
 * assigned if needed.
 * @param hello An initialization message from a router.
 */
public void processHello(Hello hello) {

  long start, finish;
  Vector interfaces;
  int node_id = INITIALZERO;
  InterfaceID myInterface;
  int bandwidth = INITIALZERO;
  IPv6Address address = new IPv6Address();
  Vector IPv6Addresses = new Vector();
  boolean newRouter = true;
  FlowRequest myFlowRequest = new FlowRequest();

  // capture the start time of processing a hello
  start = System.currentTimeMillis();

  // produce a vector of IPv6Addresses
  interfaces = hello.getInterfaceIDs();
  for (int i = INITIALZERO; i < interfaces.size(); i++){
    address = ((InterfaceID)interfaces.elementAt(i)).getIPv6();
    IPv6Addresses.addElement(address);
  }

  // check if router exists and if so, return it's node id, else return 0
  node_id = PIB.doesRouterExist(IPv6Addresses);

  // if the router does not exist in PIB
  if (node_id == ROUTERNOTINPIB){
    // assign it a new node id
    node_id = PIB.getNewNodeId();
  } else {
    newRouter = false;
```

```
}

// run through all of the LSA interfaces
for (int i = INITIALZERO; i < interfaces.size(); i++) {
  myInterface = (InterfaceID)interfaces.elementAt(i);
  address = myInterface.getIPv6();
  // if a new interface is not found in the PIB, then ...
  if (!PIB.doesInterfaceExist(address)){
    bandwidth = myInterface.getBandwidth();
    address = myInterface.getIPv6();
    // if the link is not contained in the PIB, then add it
    if (!PIB.doesLinkExist(address)){
      PIB.addLink(address, bandwidth);
    }
    // now add the interface between the node and the link
    PIB.addInterface(node_id, address);
    // now add each service level pipe
    for (int service_level = 0; service_level < numOfServiceLevels;
                                          service_level++){
      PIB.addSLP(address, service_level, INITIALDELAY, INITIALLOSSRATE,
        INITIALTHROUGHPUT);
    }
  } // end if
} //end interfaces for

// capture the hello processing finish time
finish = System.currentTimeMillis();
  gui.sendText("Server: processHello: Time required = "
    +(finish-start)+" milliseconds.");

// time since last PIB build is > 2 min and if node did not exist before
//if ((timeOfLastPIBBuild - System.currentTimeMillis()) >timeBetweenPIBBuilds
//                                          && newRouter){

  // rebuild all possible paths
  findAllPossiblePaths();

  // determine effective QoS of each path
  determineEffectiveQoSForPaths();

  // construct a new flow to this router
  try{
    myFlowRequest = new FlowRequest(IPv6Address.getByName(serverIPv6),
      address,System.currentTimeMillis(),RETURNFLOWDELAY,
      RETURNFLOWLOSSRATE,RETURNFLOWTHROUGHPUT);
  } catch(UnknownHostException uhe){
    System.err.println("Server: main: UnknownHostException: " + uhe);
   }
   processFlowRequest(myFlowRequest);
 //}

} //end processFlowRequest


/**
 * Receives link state advertisement messages from router and processes the
 * service level pipe status information that they contain. It begins by
 * checking to see if a router with the interface address described by this
 * LSA is known to the PIB. If such a router is known to exist, it then
 * checks to see if the service level pipe described by this LSA is known to
 * the PIB. If the service level pipe is known, then update its status.
 * Otherwise, add the SLP with the specified QoS characteristics. Finally,
 * update the effective QoS for the paths that pass over this service level
 * pipe by calling the determineEffectiveQoSForPaths().
 * @param router A representation of a router as defined by an LSA.
 */
public void processLSA(LinkStateAdvertisement LSA) {

 long start, finish;
 int node_id = INITIALZERO;
 int bandwidth = INITIALZERO;
 byte service_level = 0;
 int delay = INITIALZERO;
 int loss_rate = INITIALZERO;
 int utilization = INITIALZERO;
 Vector interfaces = new Vector(3,1);
 Vector SLPs = new Vector(3,1);
 IPv6Address link_id;
 IPv6Address address;
 Vector IPv6Addresses = new Vector(1,1);

 // capture the start time of processing an LSA
 start = System.currentTimeMillis();

 // produce a one element vector of IPv6Addresses
 address = LSA.getMyIPv6();
 IPv6Addresses.addElement(address);

 // check if router exists and if so, return it's node id, else return 0
 node_id = PIB.doesRouterExist(IPv6Addresses);

 // if the router does exist in PIB, then so does the interface...
```

```
  if (node_id != ROUTERNOTINPIB){

    service_level = LSA.getServiceLevel();
    delay = LSA.getDelay();
    loss_rate = LSA.getLossRate();
    utilization = LSA.getUtilization();

    if (showComments){
      gui.sendText("Server: processLSA: node_id = " + node_id
        + ", address = " + address + ", SL = "+service_level
        +", D = " +delay+ ", LR = "+loss_rate
        +", U = "+utilization);
    }

    // if this SLP is defined, then just update its status
    if(PIB.doesSLPExist(address, service_level))
    {
      PIB.updateSLP(address, service_level, delay, loss_rate, utilization);
    }
    // otherwise, insert it
    else {
      PIB.addSLP(address, service_level, delay, loss_rate, utilization);
    } // end else
  } // end if
  else { //do nothing
  }

  // capture the LSA processing finish time
  finish = System.currentTimeMillis();
  gui.sendText("Server: processLSA: Time required = "
    +(finish-start)+" milliseconds.");

  // revise the effective QoS of paths made up of this SLP
  determineEffectiveQoSForPaths(address,service_level);

} //end processLSA


/**
 * Receives and processes flow requests from applications. It begins
 * by finding a source and a destination router. These routers may be where
 * the applications are residing themselves, which is our standard situation.
 * The application could, however, reside on some host that is not registered
 * with the PIB as a router. In this case, the appropriate source or
 * destination router would be a router connected to the same link. <p>
 * The PIB is checked to ensure that there is the effective QoS available on
 * some path to satisfy the request. If a satisfactory path is found, a new
 * unique flow id is assigned and this new flow is associated with that path.
 * Each router in the path is retrieved and a new flow routing table entry is
 * sent to each. If no path can provide the requested level of QoS, then the
 * flow is assigned to zero, which will be interpreted by IPv6 as best effort
 * traffic. Finally, a flow response is sent back to the application to
 * inform it of its assigned flow id. If the flow id that is return is zero,
 * it will be the application's responsibility to either lower it QoS request
 * or to send its traffic as best effort.
 * @param flow_request The message requesting the establishment of a flow.
 */
public void processFlowRequest(FlowRequest flow_request) {

  /** A vector of slp_sequence information for a path. */
  Vector slps_in_path;
  SLPSequence currentSLPSequence,nextSLPSequence = new SLPSequence();
  int SLP_source_router, SLP_destination_router, service_level;
  IPv6Address link_id = new IPv6Address();
  IPv6Address next_hop;
  IPv6Address sourceAddress;
  int source_router, destination_router, path_id,
    flow_id=FLOWNOTSUPPORTABLE;
  long start, finish;

  // capture the start time of processing a flow request
  start = System.currentTimeMillis();

  // find a router on the same subnet as the source host
  source_router =
    PIB.findARouterOnLink(flow_request.getSourceInterface());

  // find a router on the same subnet as the destination host
  destination_router =
    (PIB.findARouterOnLink(flow_request.getDestinationInterface()));

  path_id = PIB.getPathThatCanSupportFlowRequest(source_router,
                                destination_router, flow_request);

  // if a path can support this request, then...
  if(path_id != NOSUPPORTABLEPATHINPIB){

    // assign a flow id to the request
    flow_id = PIB.getNewFlowId(path_id,source_router,destination_router,
                                            flow_request);

    lastUsedFlow_ID = flow_id;
  // determine each router in path
  // transmit Flow Routing Table Entry to it
  slps_in_path = PIB.getSLPSequenceOfPath(path_id);
  // for each router in the path, send a FRTE update
  for (int index = INITIALZERO; index < slps_in_path.size(); index++){
   // assign new slp sequence object
   currentSLPSequence = (SLPSequence)slps_in_path.elementAt(index);

   // if not the last link..
   if (index+1 != slps_in_path.size()){
    nextSLPSequence = (SLPSequence)slps_in_path.elementAt(index+1);
   }

   // retrieve values from this object
   SLP_source_router = currentSLPSequence.getSourceRouter();
   link_id = currentSLPSequence.getLinkId();
   service_level = currentSLPSequence.getServiceLevel();

   // if not the last link...
   if (index+1 != slps_in_path.size()){
    SLP_destination_router = nextSLPSequence.getSourceRouter();
   } else {
    // else it is the destination node of the flow
    SLP_destination_router = destination_router;
   }
   // determine destination address for next hop
   next_hop = PIB.getInterfaceAddress(SLP_destination_router, link_id);

   // determine source address
   sourceAddress = PIB.getInterfaceAddress(SLP_source_router, link_id);

   // send the flow routing table entry update
   sendFRTEUpdate(sourceAddress, flow_id, next_hop, service_level);

  } // end for

 } // end if

//give routers time to finish updating tables
try{
 Thread.sleep(2000);
}catch(InterruptedException ie){
 gui.sendText(ie.toString());
}
 // if the source of this flow is the server,
  if (source_router == SERVERNODEID) {
   // then add this new flow to hash table for later lookup
   if (showComments){
    gui.sendText("Server: processFlowRequest: use flow "+flow_id
     +" to send to node "+destination_router);
   }
   if (destination_router == SERVERNODEID) {
    flow_id = FLOWTOSERVER;
   }
   flowLookUp.put(new Integer(destination_router),new Integer(flow_id));
  }

  sendFlowResponse(flow_request, flow_id);

  // capture the flow request processing finish time
  finish = System.currentTimeMillis();
  gui.sendText("Server: processFlowRequest: Time required = "
   +(finish-start)+" milliseconds.");
 }

 /**
  * Receives flow termination from routers and then processes them.
  */
 public void receiveFlowTermination() { }


//***********************************************************************
// These methods handle external network communications to routers
//***********************************************************************

 /**
  * Sends a flow routing table entry update message to a router. This message
  * provides the router the required information to forward packets based on
  * its flow id.
  * @param sourceAddress The router that will receive the FRTE update.
  * @param flow_id The id assigned to the flow in question.
  * @param next_hop The IPv6 address of the next node in the path.
  * @param service_level The service level that this flow is assigned to.
  */
 public void sendFRTEUpdate(IPv6Address sourceAddress, int flow_id,
                   IPv6Address next_hop, int service_level) {
  if(showComments){
   gui.sendText("Server: sendFRTEUpdate: flowLookUp hashtable:");
   gui.sendText(""+flowLookUp);
  }
  FlowRoutingTableEntry myFRTE = new FlowRoutingTableEntry(flow_id,
                                           (byte)service_level, next_hop);
  int sourcePort = PSUEDORANDOMSOURCEPORT;
                                    //controlExec.listenToRandomPort(this);
  short destPort = ControlExecutive.SAAM_CONTROL_PORT;
  IPv6Address destHost = sourceAddress;
  // take steps to determine what flow id to send the packet on
  Vector interfaces = new Vector();
  interfaces.addElement(destHost);
  int destNodeId = PIB.doesRouterExist(interfaces);
  int flowIdToSendItOn = ((Integer)flowLookUp.get
                                    (new Integer(destNodeId))).intValue();
  try{
   controlExec.send(this,myFRTE, flowIdToSendItOn, (short)sourcePort,
                                             destHost, destPort);
  } catch (FlowException fe){
   System.err.println(fe.toString());
  }
  if (showComments){
   gui.sendText("Server: sendFRTEUpdate: FRTE for flow " + flow_id
     + " sent to interface "+sourceAddress);
   gui.sendText("          with next hop= "+next_hop
     +" on service level "+service_level+" via flow "+flowIdToSendItOn);
  }
 }


 /**
  * Sends a flow response to the requesting application to notify it of
  * its newly assigned flow id. A flow id of zero is used to indicate that the
  * flow cannot be supported. Once a flow response message is instantiated and
  * a source and destination port is defined, the control executive's send()
  * is called to send it to the destination host.
  * @param flow_request The flow request message that was received.
  * @param flow_id The flow id that is assigned to the flow request.
  */
 public void sendFlowResponse(FlowRequest flow_request, int flow_id){
  if(showComments){
   gui.sendText("Server: sendFlowResponse: flowLookUp hashtable:");
   gui.sendText(""+flowLookUp);
  }
  FlowResponse response = new FlowResponse(flow_request.getTimeStamp(),
   flow_id);
  int sourcePort = PSUEDORANDOMSOURCEPORT;
                                   //controlExec.listenToRandomPort(this);
  short destPort = ControlExecutive.SAAM_CONTROL_PORT;
  IPv6Address destHost = flow_request.getSourceInterface();
  // take steps to determine what flow id to send the packet on
  Vector interfaces = new Vector();
  interfaces.addElement(destHost);
  int destNodeId = PIB.doesRouterExist(interfaces);
  int flowIdToSendItOn = ((Integer)flowLookUp.get
                                    (new Integer(destNodeId))).intValue();
  try{
    controlExec.send(this, response, flowIdToSendItOn, (short)sourcePort,
                                                  destHost, destPort);
  }catch(FlowException fe){
    System.err.println(fe.toString());
  }
  if (showComments){
    gui.sendText("Server: sendFlowResponse: Flow response "
      + response + " from SourcePort: "+sourcePort+" to "+destHost
      + " sent via flow "+flowIdToSendItOn);
  }

 }

//******************************************************************
// These methods handle internal manipulation of data describing network status
//*******************************************************************

 /**
  * Determines all of the possible paths that exist between any source and
  * destination router in the network. This determination is based on the
  * physical definition of the network that is provided by the hello messages
  * received from the routers and stored within the PIB. The paths that are
  * found are then recorded in the PIB for fast assignment of flows later.<p>
  * All node ids are first retrieved from the PIB. For each service level, we
  * build an array of parents of each node. A parent is node that is directly
  * connected. Those directly connected nodes would have service level pipes
  * that would need to be passed through to get to the child node in question.
  * This parent array is used to populate a path table. Each node id is
  * assigned as the final destination of path and all of the different paths
  * are then found by working out from this destination. For each of these
  * destination nodes, a call is made to processPath() to find all the valid
  * paths that go to this destination node. We make the call with a specified
  * height of search of 1.
  */
 public void findAllPossiblePaths() {
  long start, finish;
  int NumberOfRouters;
  int max_slp_id = INITIALZERO;
```

```
/** A count of the highest path id assigned so far. */
int max_path_id = INITIALZERO;
int service_level = INITIALZERO;

/** A vector of the routers that are known by the db. */
Vector V = new Vector();

/** A vector of the parent routers for each given destination router. */
Hashtable parent;

// capture the start time of processing a path data
start = System.currentTimeMillis();

// reset the maximum path id assigned so far to zero
max_path_id = INITIALPATHID;

V = PIB.getAllRouterIds();

//retrieve COUNT of routers
NumberOfRouters = V.size();

//find all possible paths for each service level
max_slp_id = (new Integer(PIB.findMaxServiceLevel())).intValue();
for (service_level = INITIALZERO; service_level <= max_slp_id; service_level++){

  //build parent array of each SLP at this service level
  parent = PIB.getParents(V, service_level);

  //populate path table
  for (int index = INITIALZERO; index < NumberOfRouters; index++){
    int heightOfSearch = INITIALHEIGHTOFSEARCH;
    int aPath[] = new int[Hmax + INCREMENTATIONOFSEARCH];
    aPath[DESTINATIONNODE] = ((Integer)V.elementAt(index)).intValue();
    processPath(parent, aPath, heightOfSearch,service_level);
  }
}

// capture the path data processing finish time
finish = System.currentTimeMillis();
gui.sendText("Server: findAllPossiblePaths: Time required = "
  +(finish-start)+" milliseconds.");

timeOfLastPIBBuild = finish;

}
```

```
/**
 * Processes all valid paths that arrive at the destination node within some
 * range of hops. For each parent of the node at the distance of
 * heightOfSearch from the destination, a check is made to ensure that adding
 * this new parent will cause no cycle. If this checks out, then that parent
 * can be added and a new path can be assigned. The service level pipes in
 * this new path are identified and their sequence numbers in this path are
 * recorded to the PIB. Next, a check is made to see if the height of the
 * search is less than the server’s max search height of Hmax. If it is less,
 * the method recursively calls itself with an incremented heightOfSearch
 * variable.
 * @param parent Contains each router and a list of other
 * routers that are directly attached to them.
 * @param aPath[] An array contain a path from a source node,
 * aPath[heightOfSearch], to a destination node, aPath[0].
 * @param heightOfSearch The number of nodes in the path so far.
 * @param service_level The level of service assigned to a flow.
 */

public void processPath(Hashtable parent,
             int aPath[], int heightOfSearch, int service_level){
 IPv6Address link_id;
 int justARouter;
 int sequence_number;
 int path_id;
 Enumeration W = ((Vector)parent.get(
              new Integer(aPath[heightOfSearch-1]))).elements();
 while (W.hasMoreElements()) {
  justARouter = ((Integer)W.nextElement()).intValue();
  if (causeNoCycle(aPath, heightOfSearch, justARouter)) {

   // assign this router as the source in this path
   aPath[heightOfSearch] = justARouter;

   // record the new path id, etc.
   path_id = PIB.getNewPathId(justARouter, aPath[DESTINATIONNODE]);

   // run through the SLP’s and record their sequence
   for (int index = heightOfSearch; index >DESTINATIONNODE; index--){

    // determine link_id of this SLP
    link_id = PIB.getLinkBetween(aPath[index],
                    aPath[index-INCREMENTATIONOFSEARCH]);

    // assign the SLP its sequence number
    sequence_number = heightOfSearch - index;
```

```
        PIB.assignSLPSequence(service_level, aPath[index],
          link_id, path_id, sequence_number);
      }
      if (heightOfSearch < Hmax) {
        processPath(parent, aPath, heightOfSearch+INCREMENTATIONOFSEARCH,
                                            service_level);
      }
    }
  }
  if (showComments){
    gui.sendText("Server: processPath: paths at depth of "+heightOfSearch
      +" from node "+aPath[DESTINATIONNODE]+" is completed.");
  }
}

/**
 * Checks to ensure that the addition of a specified new node to a specified
 * path does not result in a cycle being created. This check is completed by
 * the new node is already a member of the list of nodes in the path already.
 * @param aPath[] An array contain a path from a source node,
 * aPath[heightOfSearch], to a destination node, aPath[0].
 * @param heightOfSearch The number of nodes in the path so far.
 * @param justARouter The proposed next node in for a new path.
 * @returns noCycles True if no cycles are created by the addition of
 * justARouter.
 */
public boolean causeNoCycle(int aPath[], int heightOfSearch,
                            int justARouter){
  boolean noCycles = true;
  for (int index = INITIALZERO; index < heightOfSearch; index++){
    if (justARouter == aPath[index]){
      if (showComments){
        gui.sendText("Server: causeNoCycle: adding "+justARouter
          +" to get to "+aPath[DESTINATIONNODE]+" via "
          +aPath[heightOfSearch-INCREMENTATIONOFSEARCH]
          +" at a height of "+heightOfSearch+" caused cycle!");
      }
      return noCycles = false;
    }
  }
  if (showComments){
    gui.sendText("Server: causeNoCycle: adding "+justARouter
      +" as hop #"+heightOfSearch+" to get to "+aPath[DESTINATIONNODE]
      +" via "+aPath[heightOfSearch-INCREMENTATIONOFSEARCH]
      +" does not cause cycle.");
  }
 return noCycles;
}

/**
 * Determines what the effective QoS on each path in the PIB is. For each
 * path, the service level pipes that compose it are retrieved. Then, for
 * each of these service level pipes, we total up the delay and loss rate.
 * The effective throughput remaining is determined by finding the minimum
 * difference between the observed throughput and the target throughput of
 * each service level pipe.
 */
public void determineEffectiveQoSForPaths(){
 long start, finish;
 Vector path_ids;
 Integer myPathId;
 Vector SLPs;
 SLP mySLP;
 int totalDelay = INITIALZERO, totalLossRate = INITIALZERO,
  throughput = INITIALZERO, targetThroughput = INITIALZERO,
  throughputRemaining = INITIALZERO,
  minThroughputRemaining = INITIALZERO;

 // capture the start time of processing a path data
 start = System.currentTimeMillis();

 // for each path
 path_ids = PIB.getAllPathIds();

 for (int index1 = INITIALZERO; index1 < path_ids.size(); index1++){

  // for each path
  myPathId = (Integer)path_ids.elementAt(index1);

  SLPs = PIB.getSLPsOfPath(myPathId.intValue());

  for (int index2 = INITIALZERO; index2 < SLPs.size(); index2++){

   mySLP = (SLP)SLPs.elementAt(index2);

   // add delay to total delay
   totalDelay = totalDelay + mySLP.getDelay();

   // add loss rate to total loss rate
   totalLossRate = totalLossRate + mySLP.getLossRate();

   // find min throughput
```

```
     throughput = mySLP.getThroughput();

     targetThroughput = mySLP.getTargetThroughput();

     throughputRemaining = targetThroughput - throughput;
     if (throughputRemaining < minThroughputRemaining ||
                        minThroughputRemaining == INITIALZERO){
      minThroughputRemaining = throughputRemaining;
     }

    }


   PIB.setEffectiveQoSOfPath(myPathId.intValue(),totalDelay,totalLossRate,
                                        minThroughputRemaining);
   totalDelay = INITIALZERO;
   totalLossRate = INITIALZERO;
   minThroughputRemaining = INITIALZERO;
  }

  // capture the path data processing finish time
  finish = System.currentTimeMillis();
  gui.sendText("Server: determineEffectiveQoSForPaths: Time required = "
   +(finish-start)+" milliseconds.");

 }

/**
 * Determines the effective QoS for just those paths that pass over the
 * specified service level pipe. For each path, the service level pipes that
 * compose it are retrieved. Then, for each of these service level pipes, we
 * total up the delay and loss rate. The effective throughput remaining is
 * determined by finding the minimum difference between the observed
 * throughput and the target throughput of each service level pipe.
 * @param address The address of the interface containing this service level.
 * @param service_level The service level of this SLP.
 */
public void determineEffectiveQoSForPaths(IPv6Address address, int service_level){
  long start, finish;
  Vector path_ids;
  Integer myPathId;
  Vector SLPs;
  SLP mySLP;
  int totalDelay = INITIALZERO, totalLossRate = INITIALZERO,
    throughput = INITIALZERO, targetThroughput = INITIALZERO,
    throughputRemaining = INITIALZERO, minThroughputRemaining =
INITIALZERO;

  // capture the start time of processing a path data
  start = System.currentTimeMillis();

  // for each path
  path_ids = PIB.getAllPathIdsThatTraverseSLP(address, service_level);

  for (int index1 = INITIALZERO; index1 < path_ids.size(); index1++){

   // for each link
   myPathId = (Integer)path_ids.elementAt(index1);

   SLPs = PIB.getSLPsOfPath(myPathId.intValue());

   for (int index2 = INITIALZERO; index2 < SLPs.size(); index2++){

     mySLP = (SLP)SLPs.elementAt(index2);

     // add delay to total delay
     totalDelay = totalDelay + mySLP.getDelay();

     // add loss rate to total loss rate
     totalLossRate = totalLossRate + mySLP.getLossRate();

     // find min throughput
     throughput = mySLP.getThroughput();

     targetThroughput = mySLP.getTargetThroughput();

     throughputRemaining = targetThroughput - throughput;
     if (throughputRemaining < minThroughputRemaining ||
                          minThroughputRemaining == INITIALZERO){
      minThroughputRemaining = throughputRemaining;
     }

   }


   PIB.setEffectiveQoSOfPath(myPathId.intValue(),totalDelay,totalLossRate,
                                        minThroughputRemaining);
   totalDelay = INITIALZERO;
   totalLossRate = INITIALZERO;
   minThroughputRemaining = INITIALZERO;
  }
```

```
   // capture the path data processing finish time
   finish = System.currentTimeMillis();
   gui.sendText("Server: determineEffectiveQoSForPaths: Time required ="
     +(finish-start)+" milliseconds.");
  }
  /**
   * Returns the String representation of this Server.
   * @return The String representation of this Server.
   */
  public String toString(){
   return "Server";
  } //

 //methods below are added by akkoc
/**
  * Method for receiving required values from demosation for server settings
  * Also this method is used for server to place an entry for itself
  * in the servertable
  * @return void.
  */
 public synchronized void processConfiguration (Configuration con){
  serverType = con.getServerType();
  flowId = con.getFlowId();
  metricType = con.getmetricType();
  cycleTime = con.getCycleTime();
  globalTime = con.getGlobalTime();
  AutoConfigurationExecutive ace = controlExec.getAutoConfigurationExecutive();
  ace.createNewServerInformation(flowId,controlExec.getRouterId());
  initHeartbeat();

} // end processConfigurtaion

 /**
  * Creates thread for dcm sending from the server.
  * @return void.
  */
 public void autoConfig() {
  Thread configThread = new Thread(this,"AutoConfig");
  configThread.start();
 }//end of autoconfig

 /**
  * Triggers DCM sending. and provides continues resreshment of SAAM region
  * with DCM messages.
  * @return void.
 */
public void run(){
        gui.sendText("\n Server  will send first DCM after 20 secs");
        try{
  gui.sendText("thread is sleeping now ");
  Thread.sleep(20000);
  gui.sendText("thread woke up after 20  secs so start sending ");
 }catch(InterruptedException ie){}


 while(true) {

  try{

   Vector tableEntries = controlExec.getEmulationTable().getEmTable();
   Enumeration es = tableEntries.elements();
   while( es.hasMoreElements()){
    EmulationTableEntry ent = (EmulationTableEntry) es.nextElement();
    //destination adress determined from emulationtable entry
    IPv6Address des = new IPv6Address(ent.getNextHopIPv6().getAddress());
    gui.sendText(" Destination of DCM is "+des.toString());
    byte[] nextHopBytes = des.getAddress();
    Vector interfaces = new Vector();
    interfaces = this.controlExec.getInterfaces();

    IPv6Address sInt;
    for(int i=0;i<interfaces.size();i++){
     Interface thisInterface = (Interface)interfaces.get(i);
     //cycle through all interfaces checking network address against nextHop.
     int match = 0;
     byte[] outboundInterfaceBytes = thisInterface.getID().getIPv6().getAddress();
     int bytesToCheck = 5;

     for(int index=0;index<bytesToCheck;index++){
      if((nextHopBytes[index]&0xFF)== (outboundInterfaceBytes[index]&0xFF)){
        match++;
      }//if
     }//inner for

     if(match== bytesToCheck){
      sInt = new IPv6Address(thisInterface.getID().getIPv6().getAddress());
      sendDown(sInt,des);
     }//if
    }//outer for

   }// end while
   }catch(UnknownHostException e){
    gui.sendText(e.getMessage()+"inside catch of DCM start up using em table  ");
   }//try-catch
   // added altinkaya
   // this will initiate the probing at a certain cycle we want
   // by adjusting the number of the counter variable
   if( !probeMessagesSent && (counter == 2) ){
         gui.sendText("\n Initiating server probing activation messages");
         initiateProbe();
   }//end if
   counter++;


   try{
    Thread.sleep(this.cycleTime); //from demostation
   }catch(InterruptedException ie){
    gui.sendText("thread sleep problem");
   }

}//end of while providing continues DCM sending

 }// end run()


 /**
  * Retruns flowid of server.
  * @return ind serverflow id.
  */
 public int getServerFlowId(){
  return flowId;
 }

 /**
  * Returns type of server(O-> for Primary, 1-> for Backup )
  * @return byte value.
  */
 public byte  getServerType(){
   return serverType;
 }

 /**
  * Method to send the DCM message using controlExecutive sendDCM method
  * @return void.
  */
 public void sendDown(IPv6Address srcInt,IPv6Address des) {
    DCM myDCM = new DCM(flowId,ServerId,metricType,srcInt,CTS,globalTime,
                                  getSeguenceNumberForDcmSending());
    gui.sendText("DCM with SQ is sent "+this.getSeguenceNumberForDcmSending());
    setSeguenceNumberForDcmSending();
    short sourcePort = ControlExecutive.SAAM_CONTROL_PORT;
    short destPort = ControlExecutive.SAAM_CONTROL_PORT;

    try{
      controlExec.sendDCM(this, myDCM, getServerFlowId(), sourcePort,des, destPort);
      gui.sendText("DCM has been sent");
    }catch(Exception fe){
      System.err.println(fe.toString());
    }

  }//end sendDown()

  /**
   * Method for setting proper value to put in DCM message for sequence
   * number field
   * @return void.
   */
  private void setSeguenceNumberForDcmSending(){
   sequenceNumber++;
   if(sequenceNumber == 65535) sequenceNumber = 0;
  }

  /**
  * Method for returning current sequence number value
  * @return int value.
  */

  private int getSeguenceNumberForDcmSending(){
   return  sequenceNumber;
  }


  public void processHeartbeatQuery(HeartbeatQuery hbq){
    try{

      gui.sendText("\" Heartbeat Query \" is received." + "  Seq.Num : " +
hbq.getSequenceNumber()+
                  "   at : " + System.currentTimeMillis()+"" );
      //gui.sendText(hbq.toString());
      hbr.setLastUsedFlowID(lastUsedFlow_ID);
    hbr.setSequenceNumber(hbq.getSequenceNumber());
    //test case #1
    if(testCase1 && (hbq.getSequenceNumber() == 3 || hbq.getSequenceNumber() ==
4)){

      gui.sendText("\nTEST CASE #1 " +
            "\" Heartbeat Response  \" did not send on purpose"+ "\nSeq.Num : " +
            hbr.getSequenceNumber()+"" );


    }//test case #2
    else if (testCase2 && (hbq.getSequenceNumber() == 11) ){

      gui.sendText("\nTEST CASE #2 " +
            "\" Heartbeat Response  \" did not send on purpose"+ "\nSeq.Num : " +
            hbr.getSequenceNumber()+"" );

    }//test case #2
    else if (testCase2 && (hbq.getSequenceNumber() == 12 )){

      gui.sendText("\nTEST CASE #3 " +
            "\" Heartbeat Response  \" sent with sequnce number \"11\" on purpose");
      hbr.setSequenceNumber((short)11);
      controlExec.send(this,
              hbr,
              4,
              (short)PSUEDORANDOMSOURCEPORT,
              IPv6Address.getByName("99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.1"),
              (short)8000);

    }//test case #3
    else if (testCase3 && (hbq.getSequenceNumber() > 10) &&
(hbq.getSequenceNumber() < 16)){

      gui.sendText("\nTEST CASE #3 " +
            "\" Heartbeat Response  \" did not send on purpose.  SeqNum  :"
+hbq.getSequenceNumber()+"" );


    }
    //normal case
    else{

      controlExec.send(this,
              hbr,
              4,
                    (short)PSUEDORANDOMSOURCEPORT,
                    IPv6Address.getByName("99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.1"),
                    (short)8000);


gui.sendText("*************************************************************" +
                   "\" Heartbeat Response  \" is sent"+ "  Seq.Num : " +
                   hbr.getSequenceNumber()+
                  " at : " + System.currentTimeMillis()+"" );

      }

    }
    catch(FlowException fe){
     gui.sendText(fe.getMessage());
    }
    catch(UnknownHostException uhe){
     gui.sendText(uhe.getMessage());
    }
 }



 public void processHeartbeatResponse(HeartbeatResponse hbr){

  short seqNum = hbr.getSequenceNumber();

  if(!isMainDown){


gui.sendText(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>"+
            "\" Heartbeat Response  \"  is received with"+
            " SeqNum. : " + seqNum +"  at :  "  + System.currentTimeMillis()+"");

    if(seqNum == lastSequenceNumber){
      lastResponseTime = System.currentTimeMillis();

      //gui.sendText(hbr.toString()+"\n\n");
      //gui.sendText("LastSequenceNumber :    " + lastSequenceNumber+ "");
      if(!hbController.getStatus()){
        hbController.restartQuerySendTimer();
        hbController.setStatus(true);
        //clear the vector because everything is normal again
        clearRecentMissedSequences();
        gui.sendText("vector is cleared:");
        printRecentMisses();
        }//end if

        hbController.stopResponseControlTimer();

      }
      else if (recentMissedSequences.contains(new Short(seqNum))){
        lastResponseTime = System.currentTimeMillis();
        gui.sendText("\n!!!!!! Received HeartbeatResponce Sequence Number" +
            " did not match ");
        gui.sendText("Received sequence was   :   " + hbr.getSequenceNumber()+ "");
        gui.sendText("Expected sequence was   :   " + lastSequenceNumber+ "");

        gui.sendText("Sequence Number did not match but it exists in the recent misses.
So it is accepted.");
        if(!hbController.getStatus()){
          hbController.restartQuerySendTimer();
          hbController.setStatus(true);
          //clear the vector because everything is normal again
          clearRecentMissedSequences();
          gui.sendText("vector is cleared:");
          printRecentMisses();
        }//end if

        hbController.stopResponseControlTimer();
      }
      else {

        gui.sendText("\n!!!!!! Received HeartbeatResponce Sequence Number" +
            " did not match ");
        gui.sendText("Received sequence was   :   " + hbr.getSequenceNumber()+ "");
        gui.sendText("Expected sequence was   :   " + lastSequenceNumber+ "");
        gui.sendText("Sequence Number did not match and it not in the recent misses. So
it is ignored.");
        printRecentMisses();
      }


   }//end if


  }

 /**
  * This method is added by altinkaya in March 2000 as a requirement
  * for the server probing process. This will simply display the
  * server probing result so far. For future work, the server needs
```

```
    * to compare the probing results with the LSA messages and check
    * if there is any discrepency. If any discrepency does exist then the
    * server needs to decide whether to do another probing, or just rely
    * on the fact that the probing results are more accurate.
    * @param pr The probe result message sent by the Next Node
    */
   public void processProbeResult( ProbeResult pr ){
            //this will just display the results of the Server Probing
            //future work needs to be done on the server side like what to
            //do with the results.
            gui.sendText("Probe Result Information is " + pr.toString() );
   }



   /**
    * This method is added by altinkaya in March 2000 as a requirement
    * for the server probing process. This method will initiate and send
    * send the Previous, and Next Node Activation Messages to the Previous
    * Node and the Next Node respectively. So far the probing parameters
    * are hardcoded, but can be easily changed to a more flexible format
    * by passing the intormation as parameters. The more important part of
    * the probing will be, how will the server decide to probe which link/node.
    * This is left as a future research.
    */
   public void initiateProbe(){
            //probing information
    byte typeOfProbing = 2, nicID = 0;
            int probeID = 396;
            short payloadlength = 100, measurementInterval = 500;
                        byte version = 3;
            int flowlabel = 63;
            short plength = 3;
            byte next = 3;
            byte limit = 3;

            IPv6Address source=null;
            IPv6Address destination=null;
              IPv6Header v6Header=null;

              try{
      source = new IPv6Address(
IPv6Address.getByName("99.99.99.99.0.0.0.0.0.0.0.0.0.0.0.2").getAddress());
      destination = new IPv6Address(
IPv6Address.getByName("99.99.99.99.1.0.0.0.0.0.0.0.0.0.0.2").getAddress());
```

```
    v6Header = new IPv6Header( version,flowlabel, plength, next, limit, source,
destination);
   }catch(UnknownHostException uhe){
    gui.sendText("there is problem in construction ipv6 addresses");
   }//end try-catch

  pna.setPacketFormat( typeOfProbing, probeID, v6Header,payloadlength );
  nna.setPacketFormat( typeOfProbing, probeID, v6Header, measurementInterval, nicID
);

  try{
    controlExec.send( this, pna, 1, (short)ControlExecutive.SAAM_CONTROL_PORT,
source, (short)ControlExecutive.SAAM_CONTROL_PORT );
  }catch( FlowException fe){
        gui.sendText("ERROR: " + fe.toString() );
  }//end try-catch

  try{
        controlExec.send( this, nna, 1,
(short)ControlExecutive.SAAM_CONTROL_PORT, destination,
(short)ControlExecutive.SAAM_CONTROL_PORT );
  }catch( FlowException fe){
        gui.sendText("ERROR: " + fe.toString() );
  }//end try-catch

        probeMessagesSent = true;
        }//end of initiateProbe() method



 public int getLastUsedFlowID(){

   return lastUsedFlow_ID;

 }


 public void sendHeartbeatQuery() {

   HeartbeatQuery hbq = new HeartbeatQuery();
   lastSequenceNumber = hbq.getSequenceNumber();
   try{
    controlExec.send(this,
                hbq,
                2,
                     (short)PSUEDORANDOMSOURCEPORT,
                     IPv6Address.getByName(serverIPv6),
                     (short)8000);

       currentTime = System.currentTimeMillis();
       long timeDiff =  currentTime -  lastQueryTime;
       gui.sendText("\" Heartbeat Query \" is sent with "+
                  "SeqNum. : " + lastSequenceNumber + "  at : " +
                  currentTime+"  after  :"  + timeDiff + " milliseconds");
       lastQueryTime = currentTime;
     }
     catch(FlowException fe){
       gui.sendText(fe.getMessage());
     }
     catch(UnknownHostException uhe){
       gui.sendText(uhe.getMessage());
     }

     hbq = null;
     System.gc();

     hbController.restartResponseControlTimer();

  }

  public void setIsMainDown(boolean status){

     isMainDown = status;
     if ( !isMainDown){
      bf.setVisible(false);
     }
  }

  public void addRecentMissedSequences(){

     recentMissedSequences.addElement(new Short(lastSequenceNumber));
     gui.sendText("Sequence Nubmer : " + lastSequenceNumber  + " is added to the
\"RecentMissedSequences\"vector");
     printRecentMisses();
  }
  public void clearRecentMissedSequences(){

     recentMissedSequences.removeAllElements();

  }
  public void printRecentMisses(){
    Enumeration e = recentMissedSequences.elements();
    gui.sendText("\"RecentMissedSequences\" Vector elements:  ");
    Short a;
    while (e.hasMoreElements()){

     a = (Short)e.nextElement();
     gui.sendText(""+ a.shortValue() +"");

    }

  }

  public long getLastResponseTime(){
     return lastResponseTime;
  }


  public long getLastDCMTime(){
     return lastDCMTime;
  }

  public void display(String str){
     gui.sendText(str);
  }


  private void initHeartbeat(){

   if(serverType == (byte)0){
    isMain = true;
    bf.setFrameText("THIS IS THE PRIMARY SERVER");
    bf.setVisible(true);
    gui.setTextField("The primary Server is active rigth now");
   }
   else{
    isMain = false;
    bf.setBackgroundColor(Color.cyan);
    bf.setFrameText("THIS IS THE BACKUP SERVER");
    bf.setVisible(true);
    gui.setTextField("The Backup Server is silent rigth now");
    hbController = new HeartbeatController(this, tmax, tmin, bf);
    //waits for to start sending HeartbeatQuery Messages
    Timer startHeartbeatQueryTimer = new Timer(firstQueryTime, (new
java.awt.event.ActionListener(){
                           public void actionPerformed(ActionEvent e){
                             hbController.startSending();
```

```
                        gui.sendText("querySendTimer  is started at :" +
                        System.currentTimeMillis() + "");
                 gui.sendText("First Heartbeat Query Message will be sent after  " +
                         firstQueryTime  + " milliseconds");

                      }
                     }));

     startHeartbeatQueryTimer.setRepeats(false);
     startHeartbeatQueryTimer.start();

    }//end if

  }
 }//end of class
```

REFERNCES

[1] Xie, G.G. SAAM: An Integrated Network Architecture for Integrated Services. *Proceedings of 6th IEEE/IFIP International Workshop on Quality of Service, Napa, CA,* May 1998.

[2] Xie, G.G. Efficient Management of Integrated Services Using a Path Information Base. Submitted for publication, available at http://www.cs.nps.navy.mil/people/faculty/xie/pub.html

[3] Dean Vrable, John Yarger. The SAAM Architecture: Enabling Integrated Services. Thesis, NPS, September 1999

[4] Larry L. Peterson and Bruce S. Davie. *Computer Networks, A Systems Approach.* Morgan Kaufman, 1996.

[5] Xipeng Xiao, Lionel M. Internet QoS: A Big Picture. *IEEE Network.* March/April 1999.

[6] D. Scott Alexander, Bob Braden, Carl A. Gunter, Alden W. Jackson, Angelos D. Keromytis, Gary J. Minden, David Wetherall. *Active Network Encapsulation Protocol (ANEP),* http://www.cis.upenn.edu/~swithware/ANEP/docs/ANEP.txt

[7] D.J. Wetherall, J.V. Guttag, D.L. Tennenhouse. Ants: A Toolkit for Building and Dynamically Deploying Network Protocols. *Submitted to IEEES OPENARCH '98,* 1998.

[8] Anetd: Active NETworks Daemon (v1.0) Livio Ricciulli August 10, 1998.

[9] Kevin Laik, Mary Baker. *Measuring Bandwidth.* http://mosquitonet.stanford.edu/~laik/projects/nettimer/publications/infocom1999/html/

[10] M. Hicks, P. Kakkar, Jonathan T. Moore, C. Gunter, S. Nettles. PLAN: A Packet Language for Active Networks.

[11] M. Hicks, P. Kakkar, Jonathan T. Moore. *PLAN Programmers Guide.* Available at

http://www.cis.upenn.edu/~switchware/PLAN/

[12] M. Hicks, P. Kakkar. *The PLAN Tutorial.* Available at

http://www.cis.upenn.edu/~switchware/PLAN/